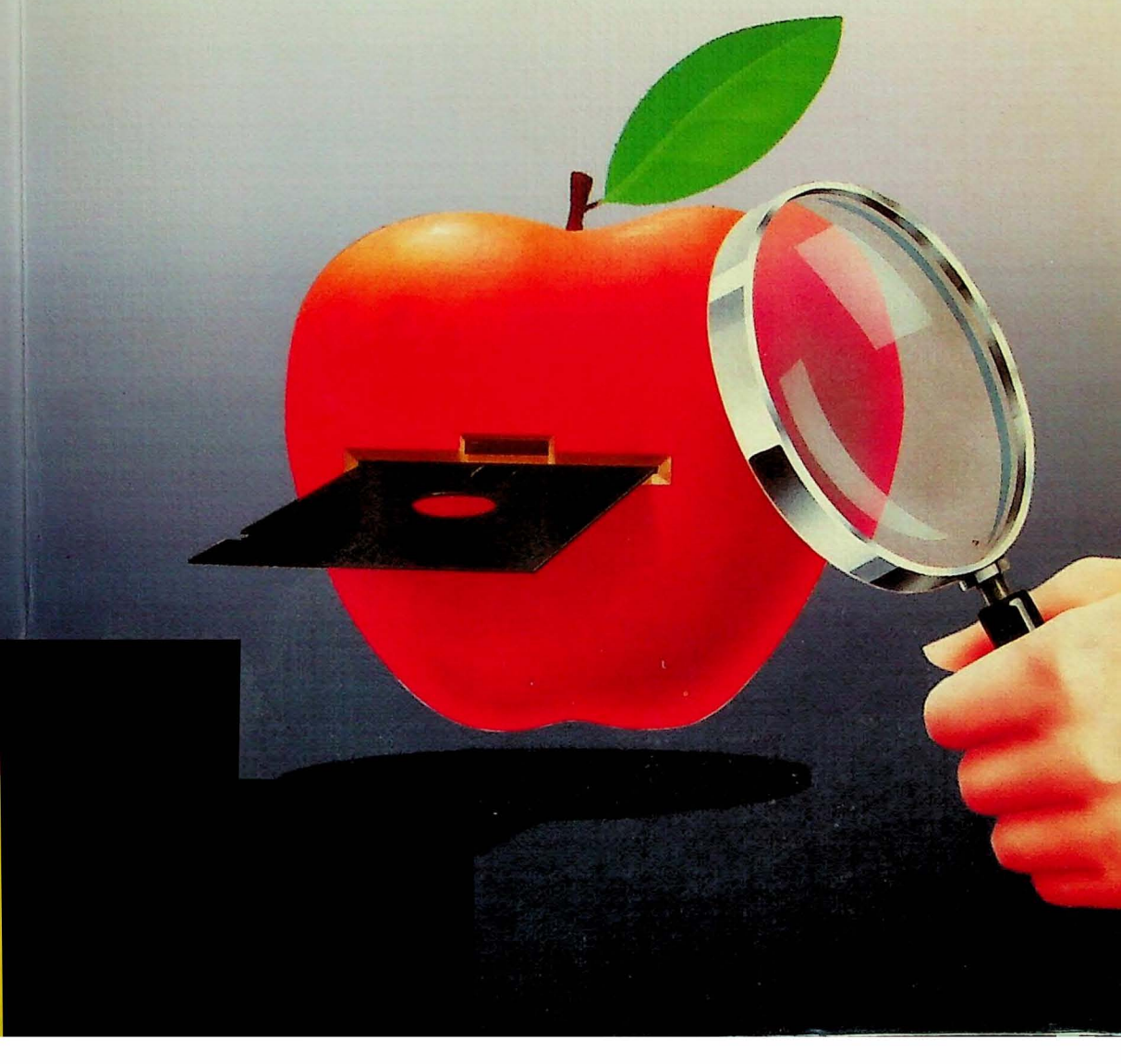
Arne Schäpers

# ProDOS-Analyse

Versionen 1.0.1, 1.0.2, 1.1.1



Arne Schäpers

# ProDOS-Analyse

**Versionen 1.0.1, 1.0.2, 1.1.1**

Dr. Alfred Hüthig Verlag Heidelberg

Diejenigen Bezeichnungen von im Buch genannten Erzeugnissen, die zugleich eingetragene Warenzeichen sind, wurden nicht besonders kenntlich gemacht. Es kann also aus dem Fehlen der Markierung ® nicht geschlossen werden, daß die Bezeichnung ein freier Warenname ist. Ebensowenig ist zu entnehmen, ob Patente oder Gebrauchsmusterschutz vorliegt.

CIP-Kurztitelaufnahme der Deutschen Bibliothek

**Schäpers, Arne:**
ProDOS-Analyse : Versionen 1.0.1, 1.0.2., 1.1.1 /
Arne Schäpers. – Heidelberg : Hüthig, 1985.
ISBN 3-7785-1134-3

© 1985 Dr. Alfred Hüthig Verlag GmbH Heidelberg
Printed in Germany
Satz und Druck: Laub GmbH & Co, Elztal-Dallau

# Vorwort

Dieses Buch wurde im Zeitraum September 84 bis Januar 85 erstellt. In der Zeit, in der die ersten Kapitel entstanden, war auf dem deutschen Markt nur die ProDOS-Version 1.0.1 verfügbar. Kurz vor Weihnachten erreichte uns die Version 1.0.2, die sich von der Version 1.0.1 nur in einigen Details unterscheidet.

Die vorliegende Analyse baut deshalb auf diesen beiden Versionen von ProDOS auf, die Version 1.1.1 erschien nach Drucklegung der ersten Kapitel. Aus diesem Grund haben die Bemerkungen über die zukünftige Weiterentwicklung der Versionen 1.0.x bestenfalls noch „historische" Bedeutung – ganz im Speziellen, was die Korrektur der 48k-Version sowie die Einsprungadresse des MLI angeht.

Wenn Sie nur über die Version 1.1.1 verfügen, lesen Sie bitte zuerst das Kapitel „Kopfstand – die Version 1.1.1", hier werden die Unterschiede zu den Versionen 1.0.x bezüglich der Speicherbelegung geklärt und aufgelistet, welche Kapitel der Teile I bis V auch für diese neue Version gültig sind.

Für seine zusätzlichen Anmerkungen zum Apple IIc möchte ich dem Lektor, Herrn Dr. J. Kehrel, meinen Dank aussprechen.

Ottobrunn, August 1985 *Arne Schäpers*

# Inhalt

## Teil I – das MLI


**1.1. Einige Vorbemerkungen** . . . . . 1

**1.2. Dateistrukturen unter ProDOS** . . . . . 4

**1.3. FCB vs. VCB/VBM und andere Interna** . . . . . 6
1.3.1. Die VBM (Volume Bit Map) . . . . . 6
1.3.2. VCB – Volume Control Block . . . . . 7
1.3.3. FCB – File Control Block . . . . . 8

**1.4. Logischer Ablauf der MLI-Befehle** . . . . . 16

**1.5. Das MLI in Blöcken** . . . . . 32

**1.6. Kommentiertes Listing des MLI** . . . . . 72

**1.7. Die Variablen des MLI** . . . . . 154

**1.8. Einige Programmiertricks im MLI** . . . . . 163
1.8.1. Die Absuch-Reihenfolge der Volumes („Drive Search Order") . . . . . 163
1.8.2. Die Behandlung des Pathname . . . . . 165


## Teil II – BOOT und REBOOT von ProDOS


**2.1. Beschreibung in logischen Blöcken** . . . . . 167
2.1.1. Das RESET-Programm im Monitor . . . . . 167
2.1.2. Das Boot-Programm des Controllers . . . . . 168
2.1.3. Der Urlader von ProDOS . . . . . 169
2.1.4. Der Relocator von ProDOS . . . . . 172

**2.2. Kommentierte Listings des Boot/Relocators** . . . . . 180
2.2.1. Das RESET-Programm im „Autostart"-Monitor . . . . . 180
2.2.2. Das Boot-Programm auf der DISKII Controller-Karte . . . . . 181
2.2.3. Der ProDOS-Urlader . . . . . 183
2.2.4. Der ProDOS-Relocator . . . . . 189

**2.3. REBOOT** . . . . . 207

## Teil III – Andere Funktionen von ProDOS

**3.1. Die Verarbeitung von NMI, RESET und IRQ** . . . . . 213

**3.2. SET TIME für die Thunderclock** . . . . . 219

## Teil IV – Die System Global Page

**4.1. Listing und Beschreibung der SGP für 64k und „128k"-Maschinen** . . . . . 222

**4.2. Unterschiede zu 48k-Systemen** . . . . . 231

## Teil V – Die RWTB der DISK II

**5.1. Die Organisation einer Diskette – Tracks und Sektoren** . . . . . 236

**5.2. Die logische Einteilung in Blocks – „Sektor-Skew"** . . . . . 239

**5.3. „Nibbilizing"/„Denibbilizing" – Lesen und Schreiben eines Sektors** . . . . . 242

**5.4. HARDWARE DETAILS – Lesen und Schreiben von Bytes und die Steuerung des Drives** . . . . . 249

**5.5. Blockbeschreibung des Drivers (RWTB) für die DISKII** . . . . . 257

**5.6. Kommentiertes Listing der DISKII-RWTB** . . . . . 269

**5.7. Die RAMDisk** . . . . . 283
5.7.1. Der Aufbau der RAMDisk und die Speicherbelegung innerhalb der AUX 64k 285
5.7.2. Blockbeschreibung des Interfaces der RAMDisk-RWTB . . . . . 286
5.7.3. Die RAMDisk-RWTB innerhalb der zusätzlichen 64k (AUX) . . . . . 287
5.7.4. Kommentierte Listings der RAMDisk-RWTB . . . . . 292
5.7.5. Das Interface zur RAMDisk-RWTB innerhalb des MLI . . . . . 295
5.7.6. Die RAMDisk-RWTB innerhalb der zusätzlichen 64k (AUX) . . . . . 297

## Teil VI – Updates von ProDOS

**6.1. Die ProDOS-Version 1.0.2** . . . . . 302
6.1.1. Änderung innerhalb von REBOOT . . . . . 302
6.1.2. Änderung innerhalb der DISKII-RWTB . . . . . 302
6.1.3. Änderungen innerhalb des MLI . . . . . 304

**6.2. Kopfstand – die ProDOS-Version 1.1.1** . . . 305
6.2.1. Die Speicherbelegung unter ProDOS 1.1.1 . . . 306
6.2.2. Die Änderungen in der Reihenfolge der vorigen Kapitel . . . 307
6.2.3. Der Boot/Relocator der Version 1.1.1 . . . 308
6.2.4. Kommentiertes Listing des RELOCATOR 1.1.1 . . . 311
6.2.5. Blockbeschreibung des MLI der Version 1.1.1 . . . 330
6.2.6. Das MLI der Version 1.1.1 . . . 332
6.2.7. Die MLI-Variablen der Version 1.1.1 . . . 415
6.2.8. Die RAMDisk-RWTB der Version 1.1.1 . . . 423
6.2.9. Interrupts der Version 1.1.1 . . . 425

## Anhänge

**Anhang A – „Spezielles“** . . . 428
1. Programmierfehler . . . 428
2. Ein Fehler in der Logik des MLI . . . 430
3. Undokumentiertes & Unklares in Reihenfolge der Speicheradressen . . . 430
4. Erwähnenswertes außerhalb des MLI . . . 433

**Anhang B – Patches für das EXER.SYSTEM und den FILER** . . . 434
1. Patch für das EXER.SYSTEM . . . 434
2. Patch für den FILER . . . 435

**Anhang C – Directory-Struktur und File-Einträge** . . . 437
1. Volume Directory . . . 439
2. Subdirectories . . . 440
3. File-Einträge . . . 441
4. „Block-Arten“ . . . 443
4.1. Der VBM-Block . . . 443
4.2. Directories . . . 443
4.3. Der Masterblock . . . 443
4.4. Der Indexblock . . . 443
4.5. Der Datenblock . . . 444
5. „Verschiedenes“ . . . 444
5.1. ACCESS . . . 444
5.2. Storage Types . . . 445

**Anhang D – Blockbelegung einer mit dem FILER formatierten Diskette** . . . 446

**Anhang E – MLI-Reference – Kurzbeschreibung der Parametersätze** . . . 449

**Anhang F – Fehlermeldungen – „normal“ und SYSTEM DEATH** . . . 459
1. „Normale“ Fehlermeldungen . . . 459
2. Fehlermeldungen von SYSTEM DEATH . . . 462

**Anhang G – Glossar der verwendeten Abkürzungen** . . . 464

**Index** . . . 466

# TEIL I – das MLI

## 1.1. Einige Vorbemerkungen

Jeder Programmierer, der ernsthaft unter DOS 3.3 gearbeitet hat, kennt den folgenden Überlegungsablauf:

a. Was soll das Programm leisten?
b. Welche DOS-Modifikationen sind nötig, damit das überhaupt funktioniert?
c. Welche Teile von DOS kann man dafür hinauswerfen?

Selbst Programme, die von Apple, Inc. selber geschrieben wurden (z.B. EDASM), kommen ohne direkte Zugriffe auf den Arbeitsbereich des File Managers nicht aus und sind deshalb in keiner Weise portabel.

Falls Sie Besitzer einer CP/M Softcard sind: Erinnern Sie sich an das ungläubige Staunen, als Sie das erste Programm von einem anderen x-beliebigen Rechner per DOWNLOAD transferiert hatten – und es funktionierte?

Ein DOWNLOAD von anderen Rechnern wird wahrscheinlich unter ProDOS nie möglich sein – es sei denn, die Hersteller anderer Computer entschlössen sich zu einer ProDOS-Implementation, die mit den in diesem Buch vorliegenden Informationen auf einem 6502-Rechner ohne größere (technische) Schwierigkeiten möglich wäre.

Was aber den oben skizzierten Überlegungsablauf betrifft – er ist unter ProDOS eigentlich nicht mehr nötig. Zumindest von der Konzeption her stellt ProDOS ein „ausgewachsenes" DOS dar. Zusammen mit dem BASIC.SYSTEM kann man zum ersten Mal von einem Betriebssystem auf dem Apple sprechen, ohne dabei gleich rote Ohren vor jedem CP/M-Benutzer zu bekommen.

In der Praxis hapert es allerdings noch an einigen Stellen, die eigentlich wenig mit dem gelieferten Code zu tun haben – Apple, Inc. schließt sich hier dem unerfreulichen Trend anderer Hersteller an. Wenn man irgendwann einmal ein Betriebssystem bekommen könnte, das nicht nur anständig funktionieren würde, sondern auch eine vollständige Dokumentation besäße – letzteres vielleicht auch noch kostenlos – es wäre nicht auszudenken!

Bevor ich jetzt weiter an meiner eigenen Existenzgrundlage säge (denn von der miesen Dokumentation der Softwarehäuser lebt unsereins schließlich), einige Gedanken und Voraussetzungen für den Leser:

## 1. Terminologie

Die Franzosen haben es sich recht einfach gemacht – Begriffe, für die eine Entsprechung in französischer Sprache existiert, wurden übersetzt, andere einfach neu geschaffen. Die deutsche Literatur zum Thema Computer schleudert seit Jahren zwischen Poly- und Univalenz, vom oft und genüßlich zitierten FPLA, der dann bei Siemens zum „Feldprogrammierbaren Gatterfeld" wird, zum „Kellerspeicherstapelrücksprungadressenzeiger", der dann im selben Satz fröhlich mit „DBRT" (Direct Beam Refresh Technology) gemischt wird. Zum Trost sei gesagt, daß die Übersetzung „Gering-Leistung-Schottky-Baustein" immer noch verpönt ist.
Bevor wir hier mit „Volumen-Bit-Mappe" anfangen, bleiben wir mit der Apple-Terminologie konsistent und verwenden die englischen Begriffe.

## 2. Voraussetzungen an den Leser

Es wäre sehr erfreulich, wenn Sie bereits „ProDOS für Aufsteiger" von Ulrich Stiehl gelesen hätten, auch wenn uns hier eine teilweise Wiederholung der dort beschriebenen Strukturen nicht erspart bleibt. Die Struktur und Bedeutung der einzelnen Bytes der Directory-Header, Dateneinträge in den Directories etc. finden Sie in Anhang C noch einmal aufgelistet .
Wenn Sie noch nie in Maschinensprache programmiert haben, möchte ich Ihnen an dieser Stelle recht herzlich für den Kauf dieses Buchs danken – stellen Sie es in Ihr Bücherregal, vielleicht können Sie einmal einem guten Freund eine Freude damit machen.

ProDOS ist – wohl aufgrund des verfügbaren Speicherplatzes – teilweise extrem „trickreich" (d.h. fast unleserlich) programmiert worden. Als abschreckendes Beispiel eine Routine, über der der Autor auch eine gute halbe Stunde gebrütet hat:

```
E330-   8A                          TXA
E331-   D0 08                 **** BNE   $E33B
E333-   AD 6E F0              *    LDA   $F06E
E336-   F0 52                 *    BEQ   $E38A  ****************
E338-   CE 6E F0              *    DEC   $F06E                 *
E33B-   CA                    ***> DEX                         *
E33C-   B1 4C              **>***> LDA   ($4C),Y               *
E33E-   91 4E              *  *    STA   ($4E),Y               *
E340-   8A                 *  *    TXA                         *
E341-   F0 1B              *  *    BEQ   $E35E ********        *
E343-   B0 2E      *************> BCS   $E373        *         *
E345-   CA         *  *  *   *    DEX    <***************      *
E346-   C8         *  *  *   *    INY                *  *      *
E347-   D0 F3      *  *  *   **** BNE   $E33C        *  *      *
E349-   A5 4D      *  *  *        LDA   $4D          *  *      *
E34B-   E6 4F      *  *  *        INC   $4F          *  *      *
E34D-   EE 6B F0   *  *  *        INC   $F06B        *  *      *
E350-   D0 03      *  *  *   **** BNE   $E355        *  *      *
E352-   EE 6C F0   *  *  *   *    INC   $F06C        *  *      *
E355-   E6 4D      *  *  *   ***> INC   $4D          *  *      *
E357-   45 4B      *  *  *        EOR   $4B          *  *      *
E359-   F0 E1      *  *  ******** BEQ   $E33C        *  *      *
E35B-   B8         *  *           CLV                *  *      *
E35C-   50 2F      *  *           BVC   $E38D  *************  *
E35E-   AD 6E F0   *  *           LDA   $F06E  <******  *  *  *
E361-   F0 1A      *  *           BEQ   $E37D  *******  *  *  *
E363-   C8         *  *           INY                *  *  *  *
E364-   D0 06      *  *           BNE   $E36C  ****  *  *  *  *
E366-   A5 4D      *  *           LDA   $4D       *  *  *  *  *
E368-   45 4B      *  *           EOR   $4B       *  *  *  *  *
E36A-   D0 03      *  *      **** BNE   $E36F     *  *  *  *  *
E36C-   CE 6E F0   *  *      *    DEC   $F06E  <***  *  *  *  *
E36F-   88         *  *      ***> DEY                *  *  *  *
E370-   4C 43 E3   *  *********** JMP   $E343        *  *  *  *
E373-   B1 4C      *************> LDA   ($4C),Y      *  *  *  *
E375-   2D 71 F0                  AND   $F071        *  *  *  *
E378-   4D 70 F0                  EOR   $F070        *  *  *  *
E37B-   D0 C8                     BNE   $E345  **********  *  *
E37D-   C8                        INY  <**************     *  *
E37E-   D0 0A                     BNE   $E38A  *******     *  *
E380-   E6E4F                     INC   $4F          *     *  *
E382-   EE 6B F0                  INC   $F06B        *     *  *
E385-   D0 03                     BNE   $E38A  ****  *     *  *
E387-   EE 6C F0                  INC   $F06C     *  *     *  *
                                                *  *     *  *
```

# 1.2. Dateistrukturen unter ProDOS

## – Ein kurzer Abriß der internen Verarbeitung

Das Konzept des „Pathname" wird an dieser Stelle als bekannt vorausgesetzt – es ist in den diversen Handbüchern ausführlich beschrieben. Zwei Anmerkungen dazu:
1. Das „Auslasten" dieses Konzepts ist natürlich nur für Festplatten sinnvoll, bei denen ansonsten bei einem CATALOG u.U. einige hundert Filenamen „vorbeirauschen" würden. Auf den Standard-Disklaufwerken von Apple führt die fünfte „sub-sub"-Directory höchstens zu unfreiwilligen Kaffeepausen und unerwarteten „VOLUME FULL"-Meldungen.
2. Das MLI selber kennt die Bezeichnungen „S6,D1" nicht. Daß entsprechende Befehle trotzdem funktionieren, liegt am BASIC.SYSTEM – der entsprechende Volumename wird eingesetzt bzw. via ONLINE vom MLI geholt.

Zu Demonstrationszwecken nehmen wir eine hypothetische Diskette mit dem Namen DEMO. CAT /DEMO zeigt folgende Einträge:

```
DEMO
NAME            TYPE        BLOCKS          MODIFIED
DATEN1          BAS         1               <NO DATE>
SUBDIR          DIR         1               <NO DATE>
BLOCKS FREE: 270            BLOCKS USED: 10
```

CAT /DEMO/SUBDIR zeigt:

```
SUBDIR
NAME            TYPE        BLOCKS          MODIFIED
DATEN2          TXT         1               <NO DATE>
BLOCKS FREE: 270            BLOCKS USED: 10
```

Wenn wir davon ausgehen, daß kein PREFIX spezifiziert ist, erfolgt der Zugriff auf DATEN1 folgendermaßen:

PATHNAME: /DEMO/DATEN1

a. Der Volume Directory Key Block wird gelesen: heißt diese Directory „DEMO" (d.h. hat dieses Volume den Namen DEMO)?
b. Der Eintrag „DATEN1" wird in diesem Block gesucht. Befindet sich dieser Name nicht in diesem Block, wird solange der nächste Directory-Block gelesen, bis die Directory zu Ende ist oder der Eintrag gefunden wird. Dieser Eintrag enthält unter anderem den Namen selber und die Blocknummer des ersten Datenblocks von DATEN1.
c. Der Pathname ist zu Ende: Der File-Eintrag wird aus dem entsprechenden Directory-Block herauskopiert und zusammen mit einigen anderen Informationen gespeichert. Die Nummer des ersten (Daten)blocks von DATEN1 ist dem System jetzt bekannt, und es kann darauf zugegriffen werden.

Der Zugriff auf /DEMO/SUBDIR gestaltet sich genauso!
Wenn wir den File DATEN2 erreichen wollen, ist ein weiterer Schritt notwendig:

PATHNAME: /DEMO/SUBDIR/DATEN2

a. Der Volume Directory Key Block wird gelesen und verglichen: heißt diese Directory DEMO?
b. Der Eintrag „SUBDIR" wird in dieser Directory gesucht.
c. Der Pathname ist noch nicht zu Ende: Der „erste Block", also der Key Block von SUBDIR, wird gelesen und es wird überprüft, ob diese Subdirectory den Namen SUBDIR hat.
d. In SUBDIR wird der Eintrag DATEN2 gesucht und die Blocknummer des ersten Datenblocks von DATEN2 zusammen mit dem „File-Info" gespeichert. Auf DATEN2 kann jetzt zugegriffen werden.
Wenn während des Zugriffs Daten verändert werden, findet nach dem CLOSE des entsprechenden Files ein Update der **gesamten** Directory-Struktur auf diesem Pathname statt. Wenn wir etwas in DATEN1 hineinschreiben und den File danach mit CLOSE (FLUSH) schließen, passiert folgendes:

a. Update-Beginn: Die Directory, in der DATEN1 verzeichnet ist, wird erneut gelesen. Der Eintrag wird herausgesucht und auf den neuesten Stand gebracht (EOF/BLOCKS USED/ STORAGE TYPE/MODIFICATION DATE & TIME). Der Directory-Block wird zurückgeschrieben.
b. Wenn wir uns nicht im Key Block der Directory befinden, wird der Key Block gelesen. Im Key Block steht der „Parent Pointer", d.h. die Nummer des Blocks der Directory, in der **diese** Directory verzeichnet ist. Da wir uns mit DATEN1 bereits im Key Block der Volume Directory befinden, ist der Prozeß damit beendet.
Schreiben wir dagegen etwas in DATEN2 hinein, befinden wir uns nun im Key Block der Subdirectory SUBDIR und erhalten als Parent Pointer die Nummer des Blocks in der Volume Directory, in dem SUBDIR verzeichnet ist. Dieser Block wird jetzt gelesen, der Eintrag SUBDIR herausgesucht und MODIFIKATION DATE & TIME verändert. Für DATEN2 ist der Prozeß damit beendet, weil sich der Eintrag SUBDIR im Key Block der Volume Directory befindet.
Für einen hypothetischen File DATEN3, der sich in einer weiteren Sub-sub-Directory befindet, geht der Prozeß mit Schritt b. solange weiter, bis wir uns im Key Block der Volume Directory befinden.

Der Zugriff auf einen noch nicht existierenden File fehlt noch in dieser Beschreibung: Die Erstellung eines neuen Files.
Der Befehl CREATE des BASIC.SYSTEM ist etwas unglücklich gewählt, MKDIR o.ä. wäre besser gewesen. So ergibt sich, daß CREATE von BASIC aus (ohne TDIR-Parameter) eine Subdirectory erstellt, während CREATE vom MLI aus vor der Erstellung **jedes** Files notwendig ist (Subdirectory oder Daten!).
CREATE läuft, was den Suchvorgang angeht, nach demselben Schema ab wie alle anderen Zugriffe – mit dem Unterschied, daß der File selber (also der letzte Teil des Pathname) nicht gefunden werden darf (DUPLICATE FILE NAME ERROR). Wird bei dem Suchvorgang ein File nicht gefunden, bleibt – logischerweise – die Directory, die als letzte im Pathname gefunden wurde (und in der der Eintrag gemacht werden soll), im Speicher.

Nach Erstellung des Eintrags wird dieser Directory-Block zurückgeschrieben mit ENTRY & UPDATE – der entsprechende Directory-Block wird gelesen und der (noch freie) Eintrag wird mit dem File-Info des neuen Files beschrieben. CREATION DATE & TIME und MODIFICATION DATE & TIME sind innerhalb des Eintrags identisch, beim Update der restlichen Directories wird nur MODIFICATION DATE & TIME übertragen.

Die letzte „Zugriffsmöglichkeit" ist DESTROY:
Nach dem Heraussuchen des File-Eintrags (Speichern des File-Info) werden die zum File gehörenden Blocks wieder freigegeben. Danach erfolgt ENTRY & UPDATE mit dem File-Eintrag, wobei hier der Storage Type des Files auf $00 („deleted") gesetzt ist, d.h. der Eintrag wird erneut herausgesucht und als „ungültig" überschrieben. Anschließend erfolgt das Update der gesamten Directory-Struktur, wobei auch hier MODIFICATION DATE & TIME bis in die Volume Directory „mitgenommen" wird – natürlich nur, falls sich der gelöschte File innerhalb einer Subdirectory befindet. Ansonsten verschwindet der Eintrag ganz, und von MODIFICATION DATE & TIME ist dann auch nicht mehr viel zu sehen.

# 1.3. FCB vs. VCB/VBM und andere Interna

Das letzte Kapitel hat eine Struktur skizziert. Aus Gründen der Übersichtlichkeit fehlt dafür aber auch fast alles, was ProDOS zum Funktionieren bringt.
Bevor wir auf Einzelheiten eingehen: Wenn Sie im Besitz des EXERCISER sind und die hier dargestellten Beispiele verfolgen wollen (sehr zu empfehlen!), lesen Sie bitte zuerst den Anhang B. „Patches". Mit dem EXER.SYSTEM ist ohne Patch leider nicht viel anzufangen: MODIFY BUFFER zeigt im MLI-Bereich das APPLESOFT-ROM.

```
Ein Memory-Layout (64k APPLE II+/e) zeigt folgende interessante Speicherbereiche:

$F600-$F7FF: $200 Byte für momentane Directory / Subdirectory
$F400-$F5FF: $200 Byte für momentane VBM – Volume Bit Map
$F300-$F4FF: $100 Byte für FCB – File Control Block (8 Einträge à $20 Byte)
$F200-$F2FF: $100 Byte für VCB – Volume Control Block (8 Einträge à $20 Byte)
$F100-$F1FF: $100 Byte für PREFIX (System Prefix & momentaner Pathname)
$F000-$F0FF: $100 Byte für Variablen des MLI
```

ProDOS unterhält über die Directory-Struktur hinaus drei weitere wichtige Verzeichnisse:

## 1.3.1. Die VBM (Volume Bit Map)

Nicht zu verwechseln mit der System Bit Map auf der Global Page – hier werden freie/belegte Blocks auf der Diskette markiert.
Jeder Block auf der Diskette wird mit einem Bit dargestellt; auf einer frisch mit dem Filer formatierten Diskette findet sich auf Block 6 folgender Inhalt:

```
01 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF .....
```

Jedes Byte umfaßt also 8 Blocks (= 8 Bit) auf der Diskette. Die Blocks sind in aufsteigender Reihenfolge verzeichnet, d.h. Byte $00 der VBM steht für die Blocks $00-$07, Byte $01 steht für die Blocks $08-$0F etc.

Innerhalb der Bytes ist die Blockreihenfolge „verkehrtherum" angeordnet, d.h. Bit 0 steht für Block $07, Bit 1 für Block $06 etc.
Auf einer frisch formatierten Diskette sind also nur die Blocks $00-$06 belegt:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | = $01 |
| Block: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | |

Ganz kurz der Inhalt dieser ersten Blocks:

Block $00: Boot für ProDOS (Track $00, Sektor $00 und Sektor $02)
Block $01: Boot für SOS – wird von der ersten Hälfte von Block $00 geladen, wenn ein APPLE III-Controller erkannt wird.
Block $02-$05: Volume Directory (4 Blocks)
Block $06: VBM
Block $07 - : Frei

Ein Block besteht aus 512 Byte. Innerhalb der VBM steht jedes Byte für 8 Blocks, d.h. ein VBM-Block faßt damit $200 * 8 = $1000 Blocks oder 2 Megabyte. Für eine normale Disk ist das weiß Gott ausreichend, für eine Harddisk (ProFile mit 5 Megabyte) langt es allerdings nicht.
Aus diesem Grund gibt es auf einer ProFile 3 VBM-Blocks, Block $07 und $08 sind damit ebenfalls belegt.
Die VBM muß bei Befehlen geändert werden, die Blocks belegen (CREATE/WRITE) oder freigeben (DESTROY/SET EOF). Die Volume Directory enthält dazu anstelle des Parent Pointer die Blocknummer des ersten (und bei Standard-Drives einzigen) VBM-Blocks.

## 1.3.2. VCB – Volume Control Block

Dieser Block existiert nur im Speicher des Apple, nicht auf der Diskette. Es können hier maximal 8 Einträge (= 8 Laufwerke) verzeichnet werden. Jeder Eintrag umfaßt $20 Byte und enthält unter anderem folgende Daten: Unitnummer, max. Anzahl der Blocks auf diesem Volume (Gesamtkapazität), Anzahl der noch freien Blocks, VBM Extent-Number (für Volumes mit mehr als einem VBM-Block), Blocknummer der VBM und die Anzahl der offenen Files auf diesem Volume.
Die Belegungsreihenfolge ist absteigend, d.h. der erste Eintrag hat die Basis $F2E0, der zweite die Basis $F2C0 etc.
Der Hauptzweck des VCB liegt in der Arbeit mit Volume-Namen (im Gegensatz zu Slotnummern). Der VCB enthält den Namen des Volumes (der im Header der Volume Directory eingetragen ist) und die Blocknummer des Key Blocks der Volume Directory. Das Setzen des Prefix beeinflußt diese Information zwar nicht – aber wenn nach SET PREFIX direkt auf eine Subdirectory zugegriffen wird, werden zusätzliche Informationen, die in der Subdirectory nicht enthalten sind (VBM-Blocknummer, Gesamtkapazität etc.) aus dem VCB geholt.
Daß dennoch das Schreiben eines Files, der in einer Subdirectory verzeichnet ist, ein gutes Stück länger dauert, liegt hauptsächlich daran, daß hier das Update der Directorystruktur komplizierter ist.

## 1.3.3. FCB – File Control Block

Dieser Block existiert ebenfalls nur im Speicher des Apple. Wenn ein File (Daten oder (Sub)directory!) mit OPEN eröffnet wird (der File muß im Gegensatz zu DOS 3.3 vorher mit CREATE erstellt worden sein!), werden hier die entsprechenden Daten für diesen File eingetragen, d.h. Blocknummer des Directory-Eintrags, Blocknummer des ersten Datenblocks, Position im File, EOF, Unitnummer des Volumes etc.).
Der FCB enthält maximal 8 Einträge (= maximal 8 offene Files).
Jeder Eintrag beansprucht $20 Byte Speicherplatz. Die Belegungsreihenfolge ist aufsteigend, d.h. der erste Eintrag hat die Basis $F300, der zweite die Basis $F320 etc.

Mit diesen Informationen können wir die Suchroutine („GET FILE") noch einmal etwas genauer verfolgen.
Das Setzen eines Prefix mit zwei Namen („/DEMO/SUBDIR") nimmt folgenden Verlauf, wenn wir davon ausgehen, daß dem MLI ein Volume namens DEMO noch nicht bekannt ist (Diskette neu eingelegt, Slot 6, Drive 1):
Der Pathname wird vom Parameterblock des „user"-Programms nach $F100 innerhalb des MLI kopiert und überprüft.
Der VCB wird nach dem Namen DEMO abgesucht, in unserem Beispiel wird DEMO nicht gefunden. Nunmehr werden alle Units (Tabelle in der Global Page) der Reihe nach behandelt – zuerst wird ein freier Eintrag im VCB gesucht, danach liest das Programm Block $0002 des jeweiligen Units (die Unitnummer ergibt sich aus der Tabelle in der Global Page).
Der Name der gefundenen Volume Directory (= Volume-Name) wird mit allen anderen Einträgen des VCB verglichen. Handelt es sich um einen bis jetzt noch nicht bekannten Namen, wird ein Eintrag gemacht. Auf diese Weise werden der Reihe nach (beginnend mit dem höchsten Slot) alle Units, falls noch nicht im VCB enthalten, verzeichnet, bis DEMO gefunden wird.
Hierbei mögliche Fehler sind VOLUME CONTROL BLOCK TABLE FULL oder VOLUME NOT FOUND, wenn DEMO auch im Header der Volume Directory des letzten Units, das in der Global Page verzeichnet ist, nicht gefunden wurde.
Vorausgesetzt, daß DEMO gefunden wurde, haben wir jetzt einen VCB-Eintrag, der folgende Daten enthält:

```
F2E0- 04                    ; Namenslänge von DEMO & „besetzt"
F2E1- 44 45 4D 4F 00 00 ... ; DEMO (max. 15 Zeichen)
F2F0- 60                    ; Unitnummer DSSS 0000, hier Drive 1, Slot 6
F2F1- 00                    ; kein offener File auf diesem Volume, sonst $80
F2F2- 18 01                 ; Gesamtzahl der Blocks ($0118 = 280)
F2F4- 0E 01                 ; Anzahl der freien Blocks ($010E)
F2F6- 02 00                 ; Blockno Key Block VOLDIR (00 02)
F2F8- xx xx                 ; nicht benutzt (00 00)
F2FA- 06 00                 ; erster VBM-Block auf 00 06
F2FC- 00                    ; VBM Extent Number 0 (1.VBM-Block)
F2FD- xx                    ; nicht benutzt (00)
F21E- 00                    ; Anzahl der offenen Files (keine)
F2FF- xx xx                 ; nicht benutzt (00)
```

Der Name „DEMO" ist damit „abgehakt", der Key Block der Volume Directory in den Speicherbereich $F600-$F7FF geladen.

Da der Pathname nach „DEMO" noch nicht zu Ende ist, wird der geladene Block jetzt nach dem Namen „SUBDIR" abgesucht. In unserem Beispiel ist das Nachladen weiterer Directory-Blocks nicht notwendig, SUBDIR steht als Eintrag im ersten Block der Volume Directory DEMO.
Die Blocknummer des ersten Datenblocks von SUBDIR steht im Eintrag selber, sie wird zusammen mit einigen anderen Informationen über SUBDIR zwischengespeichert. Damit ist die Suchroutine vorläufig beendet und kehrt zu der aufrufenden Funktion SET PREFIX zurück.
Nach der Überprüfung, ob es sich bei dem gefundenen Eintrag auch um einen Typ „Subdirectory" handelt, wird der neue Name (DEMO/SUBDIR) so kopiert, daß er genau bei $F1FF endet – der Anfangsindex des Namens in der Seite $F100 (hier $F4) wird auf der Global Page in PREFIX ($BF94) gespeichert.
Nachdem die Blocknummer des Key Blocks der Subdirectory SUBDIR auf $F060/$F061 (also **nicht** im VCB!) gespeichert worden ist, ist SET PREFIX beendet und kehrt (via Global Page) zum aufrufenden Programm zurück.

Für das nächste Beispiel gibt es zuerst wieder einmal Ärger mit der Terminologie:
OPEN von BASIC aus sucht einen File. Falls dieser File nicht gefunden wird, wird er CREATEd und danach das eigentliche OPEN ausgeführt.
Die Funktion OPEN des MLI dagegen erwartet einen existierenden File, schon deswegen, weil innerhalb des MLI mit OPEN jeder Filetyp eröffnet wird (auch Directories!). Wenn wir hier von OPEN reden, ist immer die MLI-Funktion gemeint, nicht das BASIC.SYSTEM.
Verfolgen wir den MLI-Befehl OPEN. Der Pathname ist „DATEN2", d.h. nicht vollständig. Wir befinden uns nach SET PREFIX in der Subdirectory /DEMO/SUBDIR, die Diskette befindet sich in Slot 6, Drive 1.
Zuerst wird der Pathname vom Parameterblock des „user"-Programms in den MLI-Bereich kopiert und überprüft. Hierbei wird festgestellt, daß es sich um einen unvollständigen („partial") Namen handelt. Daran schließt sich die Suchroutine an, die jetzt zuerst das Prefix des Systems abarbeitet:
Der erste Teil des Pathname (DEMO) wird im VCB gesucht (und in unserem Beispiel gefunden). Die zugeordnete Unitnummer wird aus dem VCB kopiert. Auf $F060/61 steht, solange das Prefix nicht verändert wird, die Blocknummer des Key Blocks von SUBDIR. Diese Directory ist in unserem Beispiel auch bereits die letzte Directory innerhalb des System Prefix.

Auch wenn sich die folgende Erklärung etwas in die Länge zieht: der erste Name innerhalb des Prefix dient zum Auffinden des entsprechenden VCB-Eintrags, der letzte Name muß mit der Subdirectory identisch sein, deren Blocknummer durch $F060/61 festgelegt ist. Alle Namen, die sich eventuell dazwischen befinden, sind hier uninteressant (z.B. wird bei Prefix DEMO/DIR1/DIR2/SUBDIR nur DEMO für den VCB-Eintrag und danach SUBDIR zur Verifizierung, daß $F060/61 auch wirklich auf SUBDIR zeigt (Diskettenwechsel!), gebraucht).
Der Key Block der Subdirectory (hier SUBDIR) wird geladen, es wird überprüft, ob der Header innerhalb des Blocks denselben Namen trägt wie der letzte Name innerhalb des Prefix.
Einige Werte (Gesamtzahl der Blocks, Blocknummer der VBM) werden vom VCB kopiert: Die gerade geladene Directory ist eine Subdirectory und enthält diese Informationen nicht!

SUBDIR wird nach dem Eintrag DATEN2 abgesucht, der File-Eintrag von DATEN2 wird herauskopiert („File-Info") und zusammen mit einigen anderen Informationen gespeichert. Das File-Info enthält unter anderem die Länge (EOF), ACCESS, Storage Type und die Nummer des „ersten Blocks im File".
GET FILE kehrt zur aufrufenden Funktion OPEN zurück. Jetzt erst wird der FCB nach dem Eintrag DATEN2 abgesucht.
Der Grund: Es können durchaus mehrere Files mit diesem Namen in verschiedenen Directories existieren, d.h. der Vergleich mit dem Namen „DATEN2" allein ist nicht ausreichend. GET FILE hat jetzt weitere Daten zurückgeliefert. Befindet sich ein File namens DATEN2 in dem FCB, der auf demselben Unit mit derselben Blocknummer anfängt, kann es sich nur um denselben File handeln – OPEN verabschiedet sich in diesem Fall mit ERR #50: FILE (ALREADY) OPEN.
Gleichzeitig wird ein leerer Eintrag im FCB gesucht, der dann folgendermaßen besetzt wird:

```
F300- 01          ; File Reference Number & Eintrag besetzt
F301- 60          ; Unitnummer, DSSS 0000
F302- 08 00       ; Nummer des Key Blocks der Directory, in der DATEN2
                    verzeichnet ist
F304- 08 00       ; Nummer des Blocks der Directory, in dem der Eintrag
                    DATEN2 tatsächlich steht
F306- 02          ; DATEN2 ist der 2.Eintrag innerhalb dieses Blocks
F307- 01          ; der Storage Type von DATEN2 ist 1 (nur 1 Datenblock)
F308- 00          ; STATUS. s. unten
F309- 03          ; ACCESS (0000 0011): Lesen und Schreiben erlaubt
F30A- 00          ; NEWLINE CHARACTER – nicht gesetzt
F30B- 02          ; Buffer-Index (der 1.Puffer im Global Page-Verzeichnis)
F30C- 09 00       ; der erste Block von DATEN2 ist Block 00 09
F30E- 00 00       ; Index-Blocknummer: DATEN2 hat keinen Indexblock
F310- 09 00       ; Block 00 09 befindet sich gerade im Speicher
F312- 00 00 00    ; die momentane Position innerhalb des Files
F315- 00 00 00    ; EOF des Files (dieser File hat die Länge 0)
F318- 01 00       ; BLOCKS USED: der File hat insgesamt 00 01 Blocks
F31A- xx          ; nicht benutzt
F31B- 07          ; „Level" für CLOSE und FLUSH (Kopie von $BF94)
F31C- 00          ; File (noch) nicht geändert – sonst $80
F31D- xx xx       ; nicht benutzt
F31F- 00          ; NEWLINE ENABLE MASK – nicht gesetzt
```

Einige Erklärungen dazu:

Byte $00: Die File Reference Number wird vom MLI ermittelt und dient intern eigentlich nur als „Eintrag besetzt". Ein FCB-Eintrag auf $F300 + $00 hat immer die Reference Number 1, ein Eintrag auf $F300 + $20 immer die Reference Number 2 usw. **Extern** hat diese Nummer allerdings eine zentrale Funktion: für den Namen des Files existiert im MLI kein Speicherplatz. Jeder Bezug auf einen offenen File geschieht über diese Nummer, die von OPEN zugeordnet und an das aufrufende Programm zurückgegeben wird.
Ein Beispiel für DATEN2 von BASIC aus:
OPEN DATEN2 – BASIC.SYSTEM übergibt dem MLI den Namen (+ Filetyp etc).
Das MLI bestimmt den nächsten freien Platz im FCB, führt OPEN aus und liefert die File Reference Number dem BASIC.SYSTEM. zurück.
Jeder weitere Zugriff von BASIC aus („WRITE DATEN2") wird vom BASIC.SYSTEM in „WRITE File Number 1" umgesetzt.

Bytes $02-$06: Wenn DATEN2 nicht im Key Block dieser Directory steht, sind die beiden Blocknummern verschieden. (Der Key Block wird für das Update der Directorystruktur benötigt). Die Eintragsnummer von DATEN2 wird benutzt, um bei einem CLOSE oder FLUSH den Eintrag DATEN2 innerhalb des Directory-Blocks zu finden. (Directory-Block laden – Eintragsnummer * Länge eines einzelnen Eintrags = Startadresse des Eintrags innerhalb des Blocks).

Byte $08: STATUS
Dieses Byte ist wohl das „vielseitigste" innerhalb des gesamten ProDOS. Es hat den Autor fast zur Verzweiflung gebracht. Die einzelnen Bits innerhalb von STATUS bedeuten:
Bit 7 – Wenn gesetzt, befindet sich ein modifizierter Indexblock im Speicher, der zurückgeschrieben werden muß, bevor ein neuer Indexblock geladen werden kann.
Bit 6 – Wenn gesetzt, befindet sich ein modifizierter Datenblock im Speicher, für den dasselbe gilt.
Bit 5 – Nicht benutzt (immer 0)
Bit 4 – Wird gesetzt, wenn dieser File seit OPEN verändert wurde. Dient zum Test auf „drive ready" sowie für FLUSH: Wenn Bit 6/Bit 5/Bit 4 alle gelöscht sind und Byte $1C im FCB-Eintrag („File verändert") ebenfalls nicht gesetzt ist, ist ein Update des Directory-Eintrags nicht nötig. Dies ist z.B. nach einem „nur-READ" der Fall.
Bit 3 – Wird gesetzt, wenn auf Storage Type 3 erweitert wurde. Allerdings wird dieses Bit nirgendwo getestet (s. „Spezielles").
Bit 2 – Wenn gesetzt, wird eine Erweiterung des Storage Type auf Type 3 benötigt (die maximale Länge von Type 2 wurde überschritten).
Bit 1 – Wenn gesetzt, wird eine Erweiterung auf Storage Type 2 oder ein weiterer/neuer Indexblock benötigt.
Bit 0 – Wenn gesetzt, wird ein weiterer (neuer) Datenblock benötigt.

Damit das nicht zu einfach wird, bestimmt die Anzahl der gesetzten Bits im Bereich Bit 0-2 (also 1, 2, oder 3 Bits) auch noch die Anzahl der Blocks, die für eine Erweiterung des Storage Type benötigt werden. (Type 1 => Type 2: 1 Indexblock + 1 neuer Datenblock; Type 2 => Type 3: 1 Masterblock + 1 weiterer Indexblock + 1 neuer Datenblock).

Byte $09: ACCESS
Eine Teilkopie des ACCESS-Bytes aus dem File-Eintrag: hier werden nur die letzten beiden Bits (WRITE/READ) kopiert. Der jeweilige Zugriff ist erlaubt, wenn das entsprechende Bit gesetzt ist.

Byte $0A und Byte $1F: NEWLINEs
Werden durch die Funktion NEWLINE gesetzt. Woraus auch folgt, daß NEWLINE nur bei bereits geöffneten Files funktioniert. Das BASIC.SYSTEM setzt die MASK auf $7F und den CHARACTER auf $0D, also auf <RETURN>.

Byte $0B: Buffer Index
In der Global Page existiert ein reservierter Speicherbereich für die Startadressen von 8 Puffern. Jeder offene File muß einen eigenen Puffer haben (Parameterliste des aufrufenden Programms). Der Index bezieht sich auf diese Tabelle und ist schlicht die Reference Number * 2.

Bytes $0C-$0D: Nummer des ersten Blocks
Dieser Block ist – abhängig vom Storage Type – ein Master/Index/Datenblock oder der Key Block einer Subdirectory.

Bytes $12-$14: Position innerhalb des Files (relativ!)
Hier haben wir auch gleich die Erklärung für die Differenz „max. 32 Megabyte pro Volume – max. 16 Megabyte pro File":
Für die Blocknummern sind 16 Bit reserviert, d.h. 256 * 256 Blocks à 512 Byte sind auf einem Volume möglich.
Für die Position sind 24 Bit bereitgestellt, d.h. 256 * 256 Seiten à 256 Byte.
Aus der Position läßt sich dann die dazugehörige Blocknummer sehr einfach berechnen: Position / 2 = Index innerhalb eines Indexblocks.
Beispiel: Bei Position 00 44 xx ergibt sich der $22. Block innerhalb des Indexblocks. Diese einfache Umrechnung ist wohl der Hauptgrund für die Anordnung der Blocknummern innerhalb der Indexblocks (d.h. alle niederwertigen Blocknummern auf einer Speicherseite, alle höherwertigen auf der nächsten).

Bytes $18-$19: BLOCKS USED
Diese Zahl umfaßt **alle** vom File belegten Blocks, also auch Index- und Masterblocks.

Byte $1B: „Level" ist eine Kopie von der Global Page und wird vom BASIC.SYSTEM aus gesetzt. Wann und wieso, ist momentan noch nicht ganz auszumachen.

Byte $1C: Flag für FLUSH. Wenn der File mit WRITE oder SET EOF verändert wurde, wird immer auf die Diskette zurückgeschrieben, ansonsten wird STATUS vorher getestet, ob sich noch geänderte Daten/Indexblocks im Speicher befinden.

Falls Sie nach dieser Informationsflut vergessen haben, wo wir uns in unserem Beispiel gerade befinden: OPEN setzt als nächstes eine unmögliche „Position jetzt im File" (nämlich $FF 00 00), setzt als „Position gewünscht" den Wert 00 00 00 und ruft SET MARK auf. SET MARK „erkennt", daß der gewünschte Datenblock nicht im Speicher ist und lädt ihn in den vom „user"-Programm spezifizierten Filepuffer. Wenn der File einen Storage Type > 1 hat, wird vorher der entsprechende Indexblock (in die höheren beiden Seiten) des Puffers geladen. OPEN setzt als letztes die File Reference Number in den Parameterblock des „user"-Programms ein und kehrt dann zurück.

Schreiben wir etwas in DATEN2 hinein:
Das MLI „holt" sich die beiden Parameter des WRITE-Befehls, die Reference Number und den „Request count", also die Anzahl der zu schreibenden Bytes. Das Suchen des Files auf der Diskette entfällt, der FCB enthält ja bereits die nötigen Daten.
Die momentane Position + Request count ergeben die benötigte Mindestlänge des Files DATEN2. In unserem Fall ist die momentane Länge von DATEN2 gleich Null, d.h. jede Schreibaktion überschreitet den „alten" EOF mit Sicherheit.
Wenn innerhalb von WRITE der EOF eines Files überschritten wird, wird die Funktion SET EOF aufgerufen, die einfach das Ergebnis von „Position jetzt" + Request count im FCB als neuen EOF speichert. Der „alte" EOF wird (für SET MARK) zwischengespeichert.

Ein BASIC-Beispielprogramm dazu:

```
10 PRINT CHR$(4) „OPEN DATEN2"
20 PRINT CHR$(4) „WRITE DATEN2" : PRINT „XXXXXX"
```

Dieses Programm wird nach einem RUN mit der Meldung FILE(S) OPEN enden. Die beiden Instruktionen WRITE und PRINT stehen absichtlich auf einer Zeile: „WRITE DATEN2" ist – schon wieder Ärger mit der Terminologie – von BASIC aus **kein** MLI-Befehl, er setzt lediglich den Status innerhalb des BASIC.SYSTEM. Erst wenn danach mit PRINT Daten dazukommen, wird die Länge dieser Daten berechnet und das MLI mit dem entsprechenden Request count aufgerufen.
WRITE innerhalb des MLI ruft, nachdem der neue EOF (falls nötig) gesetzt wurde, SET MARK auf.
SET MARK (Setze Position im File) bekommt in unserem Beispiel als gewünschte Position den Wert 00 00 00 übergeben, der noch durch OPEN gesetzt wurde. Dieser Block befindet sich bereits im Speicher. Darum passiert hier erst einmal gar nichts.
WRITE kopiert jetzt die vom BASIC.SYSTEM übergebenen Daten in den Puffer, der bei OPEN zugeordnet wurde und setzt „Geänderter Datenblock im Speicher" im FCB (STATUS, Byte $08, Bit 6) , „File seit OPEN geändert" (STATUS, Bit 4) und „File geändert" (Byte $1C, Bit 7).
Wie man sieht, ist auch unter ProDOS das Schreiben von Textfiles eine überaus „mühselige" Angelegenheit:
– das BASIC.SYSTEM übergibt dem MLI die Startadresse der Daten und die Anzahl der zu schreibenden Bytes;
– das MLI kopiert diese Daten in den dem File zugeordneten Puffer;
– und (endlich!) wird dieser Puffer „vernibbelt", d.h. in für die Disk verdauliche Bytes umgesetzt und geschrieben.

Eine Anmerkung für an DOS gewöhnte Programmierer: Der einzige strukturelle Unterschied zu DOS besteht hier in der Art und Weise, wie die Nibbles erzeugt und geschrieben werden. DOS hat den gesamten Puffer während des Prenibble-Prozesses noch einmal kopiert (nach $BB00-$BC55) und von da aus geschrieben. ProDOS berechnet nur die „zusätzlich" benötigten Bits in einem eigenen Puffer, der damit nur $56 Byte lang ist, und schreibt die restlichen Daten direkt vom Filepuffer aus. Genauere Informationen dazu finden Sie in Teil V – RWTB.

Zurück zu WRITE: Wird bei der Kopieraktion die Länge eines Datenblocks nicht überschritten, endet WRITE hier. Die Anzahl der Bytes als „Transfer count" wird im Parameterblock des aufrufenden Programms gespeichert, der Datenblock verbleibt im Filepuffer und wird erst dann auf die Diskette geschrieben, wenn ein neuer Datenblock in den Puffer geladen werden muß oder der File geFLUSHt wird. (CLOSE ruft ebenfalls FLUSH auf).
Der FCB-Eintrag hat ein paar Änderungen erfahren:
Bei unserem Beispiel stehen „Position jetzt" und EOF nun auf 00 00 06, im STATUS-Byte sind die Flags „Geänderter Datenblock im Speicher" und „File seit OPEN geändert" gesetzt, das Flag „File geändert" (Byte $1C, Bit 7) ist ebenfalls gesetzt.
Wenn Sie das BASIC-Programm laufen lassen, werden Sie feststellen, daß der Diskettenmotor trotzdem kurz anspringt, weil vor dem **ersten** Verändern eines Files seit OPEN auf „drive ready" getestet wird. Die entsprechende Routine wird nur aufgerufen, wenn Bit 4 in

STATUS noch nicht gesetzt ist, d.h. nur einmal. Bei einem zweiten WRITE in den File bleibt die Disk ruhig, solange dabei keine Blockgrenze überschritten wird.
Wir fügen dem BASIC-Beispielprogramm eine weitere Zeile hinzu:

```
30 FOR X = 1 TO 510 : PRINT „X";: NEXT
```

Jetzt wird es schon etwas interessanter, weil wir damit die Blockgrenze (und das Fassungsvermögen des Storage Type 1) überschreiten.
WRITE ruft wieder GET MARK mit der gewünschten Position auf. Vor dem Start der Schleife ist diese Position unverändert, nämlich 00 00 06. Der entsprechende Block ist im Speicher, SET MARK kehrt also zurück, ohne etwas zu unternehmen. Die Kopier-Routine wird aufgerufen und bricht ab, wenn das Blockende überschritten wird. Die „Position jetzt im File" ist während des Kopiervorgangs laufend erhöht worden und steht jetzt auf 00 02 00, also am Anfang des nächsten Blocks. GET MARK wird wiederum aufgerufen und stellt dieses Mal fest, daß der gewünschte Block zur Position 00 02 00, also der zweite Block des Files, nicht im Speicher ist. Bevor GET MARK versucht, diesen Block zu lesen, wird Bit 6 von Byte $08 im FCB getestet („Veränderter Datenblock im Speicher"). Da der Test hier positiv ausfällt, wird der Datenblock auf die Diskette zurückgeschrieben.
GET MARK überprüft jetzt die gewünschte Position und stellt dabei fest, daß das Fassungsvermögen von Storage Type 1 überschritten wurde. Bit 0 und Bit 1 in STATUS des FCB werden gesetzt (Erweiterung auf Type 2 – neuer Indexblock/weiterer Datenblock benötigt). Daraufhin wird sowohl die untere Hälfte des Datenpuffers (zukünftiger Datenblock) als auch die obere Hälfte (zukünftiger Indexblock) gelöscht. SET MARK kehrt zu WRITE zurück.
WRITE testet STATUS und stellt fest, daß eine Erweiterung des Storage Type notwendig ist. Aus der Anzahl der gesetzten Bits im Bereich der Bits 0-2 von Byte $08 ergibt sich der Bedarf an Blocks, um diesen File weiterschreiben zu können: Ein Indexblock sowie ein weiterer Datenblock werden benötigt. Die Routine „Space on Volume?" wird aufgerufen, um zu überprüfen, ob noch mindestens diese zwei Blocks auf der Diskette frei sind, d.h. im Falle eines Falles erfolgt der Abbruch mit „VOLUME FULL" **vor** der Erweiterung. Der nächste freie Datenblock wird bestimmt und in der VBM belegt.
Der folgende „Umbauprozeß" ist etwas kompliziert:
Im FCB stehen drei Blocknummern: „erster Block des Files" (Bytes $0C/$0D), „Indexblock momentan im Speicher" (Bytes $0E/$0F) und „Datenblock momentan im Speicher" (Bytes ($10/$11).
Die Nummer des „ersten Blocks" landet im zukünftigen Indexblock (obere Hälfte des Puffers) an erster Stelle. Die Blocknummer des „nächsten freien Blocks" wird auf „erster Block" und auf „Indexblock momentan im Speicher" kopiert. Der (über)nächste freie Block wird bestimmt und belegt, die Blocknummer landet auf „Datenblock momentan im Speicher" und als zweiter Eintrag im Indexblock.
In STATUS wird Bit 7 (veränderter Indexblock im Speicher) gesetzt. Der Storage Type des Files (Byte $07 im FCB) wird um eins erhöht.
Als Ergebnis haben wir jetzt einen Indexblock als ersten Block des Files, der momentan auf zwei Datenblocks zeigt: den vorherigen „ersten Block" und einen freien Block auf der Diskette, dessen Inhalt uns nicht zu kümmern braucht, er wird dann mit dem Pufferinhalt überschrieben.

Bei einem späteren CLOSE wird auch der Inhalt des Directory-Eintrags (hier speziell „erster Block des Files" und Storage Type) auf den neuesten Stand gebracht. Die Erweiterung des Storage Type ist damit beendet.
WRITE kopiert jetzt noch die letzten von unserem BASIC-Programm generierten Zeichen. Da der neue Datenblock dabei nicht voll wird, bleibt er erst einmal im Speicher (Bit 4 von STATUS wird wieder gesetzt).
Das Vergrößern des Files ist für die nächsten 254 Blocks sehr einfach: Wenn eine Blockgrenze überschritten wird, ruft WRITE die Routine SET MARK mit der Position $00 im nächsten Block auf. SET MARK stellt fest, daß dieser Block nicht im Speicher ist, schreibt den jetzigen Block (aufgrund des gesetzten Bit 4 von STATUS) auf die Diskette zurück, setzt Bit 0 von STATUS (neuer Block benötigt) und löscht die untere Hälfte des Puffers (Datenblock). WRITE ruft ALLOCATE NEXT BLOCK (suche nächsten freien Block und belege ihn) auf und trägt die gefundene Blocknummer im Indexblock ein, der sich immer noch in der oberen Hälfte des Filepuffers befindet.
Die Erweiterung von Storage Type 2 auf Type 3 geschieht fast genauso wie die Erweiterung von Type 1 auf Type 2, nur daß dieses Mal der „erste Block des Files", der momentan ein Indexblock ist, durch einen Masterblock ersetzt wird, der seinerseits als ersten Eintrag die Nummer des Indexblocks enthält. Als zweiten Eintrag bekommt dieser Masterblock die Nummer des nächsten freien Blocks auf der Diskette. Dieser Block ist der nächste Indexblock des Files und wird auch nicht gelesen, sondern nur in der VBM belegt, im Speicher erst einmal gelöscht und später auf die Diskette geschrieben.

Wenn Sie jetzt genau aufgepaßt haben, werden Sie feststellen: Der Puffer umfaßt $400 Byte für jeden File, also Raum für einen Daten- und einen Indexblock. Wo ist der Platz für einen Masterblock?
Ganz einfach: es gibt auch keinen Platz dafür.
Ein Masterblock ist immer nur temporär im Speicher, und zwar auf dem Platz des Indexblocks. Falls der Indexblock verändert wurde, ist er vorher zurückgeschrieben worden (WRITE/SET EOF) bzw. zu Ende gelesen (READ).
SET MARK liest den Masterblock in den Indexteil des Filepuffers, bestimmt die Blocknummer des Indexblocks über die Position, löscht danach entweder den Indexteil des Filepuffers („neuer Indexblock") oder überschreibt ihn durch das Lesen des Indexblocks vom Volume („Indexblock existiert").

Schließen wir den Beispiel-File:
CLOSE hat nur einen Parameter, die File Reference Number. Abgesehen von CLOSE (Reference Number Null = ALL), wo der gesamte FCB abgearbeitet wird, bestimmt CLOSE über diese Nummer den entsprechenden Basis-Index im FCB und ruft dann als erstes FLUSH auf. Wurde während WRITE der vorherige EOF des Files überschritten, wird grundsätzlich geFLUSHt, ansonsten stellt FLUSH anhand von STATUS fest, ob sich momentan veränderte Daten/Indexblocks im Speicher befinden und schreibt sie gegebenenfalls auf die Diskette.
Als nächstes kopiert FLUSH die entsprechenden Parameter aus dem FCB, die auch im Directory-Eintrag enthalten sind (Storage Type, „erster Block" und EOF) sowie die Daten, wo der Eintrag für diesen File zu finden ist (Directory-Blocknummer, Eintragsnummer innerhalb des Blocks), sucht den File-Eintrag heraus und überschreibt die entsprechenden Parameter mit den neuen Daten. Danach ist wieder ein kompletter File-Eintrag vorhanden, der dann via ENTRY & UPDATE den alten File-Eintrag überschreibt. ENTRY & UP-

DATE lädt die entsprechende Directory, berechnet anhand der Eintragsnummer die Startadresse des Eintrags selber und überschreibt ihn mit dem vorbereiteten Parametersatz („File-Info"). Danach arbeitet sich diese Routine bis zur Volume-Directory „hoch" (die Anzahl der belegten/freien Blocks sowie „Modification DATE & TIME" werden „mitgenommen") und kehrt zu FLUSH zurück. FLUSH schreibt die VBM zurück (falls die VBM dieses Units noch im Speicher steht, diesen Prozeß habe ich Ihnen bisher unterschlagen) und kehrt zu CLOSE zurück.
Der zugeordnete Puffer wird aus der System Bit Map entfernt, d.h. der Speicherplatz wird wieder freigegeben und der FCB-Eintrag durch eine Null auf Byte $00 ungültig gemacht. Über die Unitnummer des Eintrags wird der VCB-Eintrag herausgesucht und die Anzahl der offenen Files auf diesem Unit (Byte $1E) um eins vermindert. Bleibt dabei kein offener File mehr übrig, wird das „OPEN File(s) on Unit"-Flag im VCB (Byte $11) ebenfalls zurückgesetzt. Damit ist unser File CLOSEd und dieser Parforce-Ritt durch ProDOS ebenfalls.

# 1.4. Logischer Ablauf der MLI-Befehle

## Einsprung und Command Interpreter

Die Startadresse des MLI für die ProDOS-Version 1.0.1 ist $D000. Apple, Inc. hat sich mit dem Einsprung über die Global Page ($BF00) die Möglichkeit offengehalten, diese Adresse in zukünftigen Versionen zu verändern (bei Version 1.1.1 bereits geschehen!).
Der Einsprung erfolgt via JMP, nicht als Subroutine.
X- und Y-Register werden auf der Global Page gespeichert. Die Returnadresse zum aufrufenden Programm („user") wird vom Stack geholt und als Pointer in der Zeropage gespeichert, um 4 erhöht und als MLIRTS auf der Global Page abgelegt. SYSERR wird auf Null gesetzt.
Über die Returnadresse des aufrufenden Programms kommen wir zum **COMMAND-Byte**. Aus dem COMMAND wird ein Index im Bereich $00-$1F berechnet. Über diesen Index wird das COMMAND mit einer Tabelle verglichen, in der alle Befehls-Bytes verzeichnet sind. Stimmt COMMAND nicht mit dem entsprechenden Tabellen-Byte überein, folgt ERR#1: BAD SYSTEM CALL NUMBER.
Eine zweite Tabelle enthält (mit demselben Index) die Anzahl der Parameter. Hier wird ebenfalls verglichen. Bei Nichtübereinstimmung folgt ERR#4: BAD SYSTEM CALL PARAMETER COUNT.
An dieser Stelle wird auf den ersten nachträglich eingebauten Befehl geprüft:
**$82– (SET TIME)** ist das einzige Command ohne Parameter (d.h. Parameter Count = 0) und bewirkt einen Aufruf von $BF06, wo dann ein Vektor zur Uhr-Routine stehen sollte. Fehlt die Uhrenkarte, steht hier ein RTS. In beiden Fällen wird das MLI nach diesem Aufruf via $D078 verlassen.
Als nächstes wird auf den zweiten nachträglich eingeflickten Befehl hin verglichen:
**$65– (REBOOT)**. Dieser Befehl springt (JMP!) zu $BF03 (und von da aus allerdings wieder in das MLI).

Durch Links-shiften des COMMAND verteilt sich die Befehlsausführung auf drei Gruppen: $4x, $8x, $C0-$D3.

Alle Befehle dieser drei Gruppen werden via JSR aufgerufen und enden auf $D078 (Ausnahme: SYSTEM DEATH bei Befehlsausführung).

Bei $D078 werden X- und Y-Register wieder zurückgeladen, (A enthält SYSERR) und das MLI wird via INTEXIT ($BFA0) verlassen.

**$4x Command Handler**

Hier wird nur unterschieden, ob $40 (ALLOCATE INT) oder $41 (DEALLOCATE INT). Danach beginnt die eigentliche Befehlsausführung, d.h. Holen der Parameter und Belegen bzw. Löschen der Interrupt-Startadressen in der Global Page. Nach Ausführung: RTS => JMP $D078.

**$8x Command Handler**

Die Parameter des aufrufenden Programms werden auf die Zeropage kopiert. Die benötigte Seitenzahl des Puffers (bei Start auf Seitengrenze, also $xx00: 2 Seiten, ansonsten 3 Seiten) wird berechnet und der Speicherbereich des Puffers mit der System Bit Map verglichen. Erstaunlicherweise wird dieser Vergleich für READ **und** WRITE durchgeführt, d.h. auch WRITE ist über dieses Kommando für in der Bit Map belegte Speicherseiten nicht möglich und endet mit ERR#56: BAD BUFFER ADDRESS.
Allerdings wird bei diesem Vergleich nicht geprüft, ob die Zieladresse oberhalb von $BFFF liegt (die System Bit Map ist hier zu Ende) – entsprechende Versuche zeitigen erstaunliche Ergebnisse (s. „Spezielles").
Sind die entsprechenden Speicherseiten in der System Bit Map frei, wird über die Unitnummer die entsprechende RWTB (Read/Write Track/Block) aufgerufen. Die Unterscheidung READ/WRITE findet innerhalb der RWTB statt. Nach Ausführung: RTS => JMP $D078.

**$C0-$D3 Command Handler**

Dieser Command Handler wird von $D075 mit JSR aufgerufen und kehrt damit nach Ausführung direkt zu $D078 zurück.
Der aus COMMAND gewonnene Index wird auf den Bereich $00-$13 reduziert und damit eine weitere Tabelle ausgewertet, die folgendes Format hat:
Bit7– Wenn gesetzt, Aufruf von Copy & Verify Pathname ($D27F);
Bit6– Wenn gesetzt, Aufruf von Set File Data ($D3C5);
Bit5– Wenn gesetzt, Aufruf von SET TIME ($BF06);
Bit4-Bit0: Index in der nächsten (vorläufig letzten) Tabelle, die dann die Startadressen der eigentlichen Befehle enthält.

Die drei oben angeführten Routinen werden **vor** den eigentlichen Befehlen ausgeführt. Es ergeben sich folgende Abläufe:

| COMMAND | $D27F | $D3C5 | $BF06 | Startadresse |
|---|---|---|---|---|
| $C0- CREATE | X | | X | $D4F7 |
| $C1- DESTROY | X | | X | $EB50 |
| $C2- RENAME | X | | X | $EA0B |
| $C3- SET FILE INFO | X | | X | $E9D4 |
| $C4- GET FILE INFO | X | | | $E96F |
| $C5- ONLINE | | ---------- | | $D42A |
| $C6- SET PREFIX | | ---------- | | $D32A |
| $C7- GET PREFIX | | ---------- | | $D382 |
| $C8- OPEN | X | | X | $E0B9 |
| $C9- NEWLINE | | X | | $E95D |
| $CA- READ | | X | | $E1F7 |
| $CB- WRITE | | X | | $E47C |
| $CC- CLOSE | | X | | $E677 |
| $CD- FLUSH | | | X | $E6E8 |
| $CE- SET MARK | | X | | $DE1E |
| $CF- GET MARK | | X | | $DE08 |
| $D0- SET EOF | | X | | $E807 |
| $D1- GET EOF | | X | | $E94B |
| $D2- SET BUF | | X | | $EEA7 |
| $D3- SET BUF | | X | | $EE98 |

Bei den meisten Befehlen ist der Grund, warum diese Routinen vorher aufgerufen werden, offensichtlich:
Alle Befehle, die sich auf einen Pathname beziehen, benutzen Copy & Verify Pathname. Ausnahme: SET PREFIX. Hier wird diese Routine innerhalb der Befehlsausführung selber aufgerufen. Grund: SET PREFIX ist mit einem Nullargument („PREFIX /") möglich, und Copy & Verify Pathname würde hier mit einem Fehler abbrechen. Die Routine wird innerhalb von SET PREFIX aufgerufen, wo dieser Fehler abgefangen werden kann.

Alle Befehle, die sich auf einen File via Reference Number beziehen, rufen Get File Data auf. Ausnahme: CLOSE und FLUSH, weil hier auch mit Reference Number 0 (= ALL) gearbeitet werden kann.

Schlußendlich: Alle Befehle, die Files erstellen (CREATE), löschen (DESTROY) oder nach erfolgter Veränderung wieder schließen (CLOSE/FLUSH), versuchen DATE & TIME vom System zu erhalten. Eine Ausnahme bildet hier SET FILE INFO: Die Systemzeit wird zwar abgefragt, aber nur tatsächlich eingesetzt, wenn MOD DATE & TIME innerhalb des Parameterblocks nicht eigens gesetzt sind.

An dieser Stelle noch eine Anmerkung zur SET TIME-Funktion:
**SET TIME** setzt nur die entsprechenden Speicherstellen in der Global Page auf den neuesten Wert, der dann von den einzelnen Befehlen des MLI abgefragt werden kann. Eine Datenübertragung in den MLI-Bereich selber findet bei SET TIME nicht statt.

**$C0– CREATE (Start $D4F7)**

Vorher wurde Copy & Verify Pathname sowie SET TIME ausgeführt.

Der File wird gesucht (FIND FILE: $D7AB). Wenn gefunden, ERR#47: DUPLICATE FILE NAME.
Der Storage Type wird aus dem Parameterblock des „user"-Programms geholt und verifiziert (0-3 oder $0D für Subdirectory).
Test auf „drive ready" ($E65B).
Wenn die Directory, die als letztes im Pathname gefunden wurde (und in die der Eintrag des neuen Files gemacht werden soll), keinen freien Eintrag mehr hat:
$D524– Test, ob Volume- oder Subdirectory. Wenn Volume Directory, ERR#49: VOLUME DIRECTORY FULL. Ansonsten ALLOCATE NEXT BLOCK ($DCA9); der Block wird an die Subdirectory angehängt und CREATE von neuem gestartet.
$D5B6– Freier Eintrag gefunden: Der Directory Space ($F600-$F7FF) wird gelöscht ($D5A9), die entsprechenden Daten vom Parameterblock des „user"-Programms werden in den Variablenbereich kopiert; ALLOCATE NEXT BLOCK ($DCA9).
Wenn der neue File eine Subdirectory ist, wird ein Subdirectory Header in den Directory Space kopiert, ansonsten bleibt dieser Raum leer.
Der Block wird auf den von ALLOCATE bestimmten und belegten freien Platz auf der Diskette geschrieben.
Die Ausführung geht mit ENTRY & UPDATE ($D6AB) weiter. Die Directory, in der der neue Eintrag gemacht werden soll, wird geladen und die vorher gesetzten Parameter (d.h. der gesamte Eintrag) geschrieben. Das Update endet in der Volume Directory und kehrt zurück (hier zu $D078).

**$C1– DESTROY (Start: $EB50)**

Vorher wurde Copy & Verify Pathname sowie SET TIME ausgeführt.

Der File-Eintrag wird gesucht und geladen (GET FILE: $D798).
Der File wird im FCB gesucht ($E1A9). Wenn gefunden, ERR#50: FILE OPEN.
Aufruf von „Space on Volume?", um die Anzahl der freien Blocks auf dem Volume zu bestimmen; diese Anzahl wird nach DESTROY entsprechend erhöht.
Test des ACCESS des Files. Wenn DESTROY nicht erlaubt („LOCKED"), Exit via SET SYSERR ($BF09) mit ERR#4E: FILE ACCESS ERROR.
Test auf „drive ready" ($E65B). Test des Storage Type.

$EBA0– File ist Datenfile: der Grenzwert (neuer EOF) des Files wird auf $000200, d.h. 1 Block gesetzt und REMOVE BLOCKS ($EC62) aufgerufen. Hier wird der Storage Type des Files stufenweise bis auf Type 1 heruntergesetzt, die Blocks werden in der VBM wieder freigegeben. Zum Schluß wird auch noch der jetzt „erste Block im File" entfernt. Sprung zu

ENTRY & UPDATE ($D6AB), wo der File-Eintrag mit Storage Type 0 neu geschrieben wird. RTS zu $D078 von ENTRY & UPDATE aus.

$EBF8– File ist Subdirectory:
Zuerst wird festgestellt, ob diese Subdirectory leer ist („Active Entries" = 0), ansonsten ERR#4E: FILE ACCESS ERROR.
Namenslänge/Storage Type im Key Block (Byte $00) wird auf Null gesetzt und der Key Block zurückgeschrieben. Über die Forward Reference wird der nächste Directoryblock geladen (nur aufgrund weiterer Forward References) und via DEALLOCATE BLOCK ($DC27) in der VBM freigegeben. Diese weiteren Directoryblocks werden nicht mehr zurückgeschrieben. Die Schleife geht solange, bis die gefundene Forward Reference gleich 00 00 ist, d.h. der letzte Directory-Block in der VBM wieder freigegeben ist. Danach wird als letztes der Key Block selber freigegeben und DESTROY endet mit ENTRY & UPDATE ($D6AB), wobei der File-Eintrag für die Subdirectory mit Storage Type 0 neu geschrieben wird. RTS zu $D078 von ENTRY & UPDATE aus.

**$C2– RENAME (Start: $EA0B)**

Vorher wurde Copy & Verify Pathname sowie SET TIME aufgerufen.

Der File-Eintrag des Originalfiles wird gesucht und geladen (FIND FILE ENTRY: $D7AB). Dabei kommt die Suchroutine mit ERR#40 (INVALID PATHNAME) zurück, wenn nur ein Volume-Namen angegeben ist. Anhand dieses „Fehlers" wird unterschieden, ob es sich um das RENAME eines Volumes oder einer Subdirectory/eines Files handelt.

$EA14– RENAME Volumes:
Der Pointer zum neuen Namen wird aus dem „user"-Parameterblock geholt, der neue Name kopiert und verifiziert ($EB35). Der Path zum neuen Namen darf ebenfalls nur aus einem einzigen Namen bestehen. Falls auf dem Volume ein File OPEN ist (Flag im VCB-Eintrag), folgt ERR#50: FILE IS OPEN.
Der neue Volume-Namen wird um „Storage Type $0F" (= Volume Directory) bereichert. Danach wird er in den durch GET FILE geladenen Key Block der Volume Directory kopiert und der Key Block wird zurückgeschrieben ($EB26). Im entsprechenden VCB-Eintrag wird der alte Name ebenfalls mit dem neuen Namen überschrieben. RENAME (Volumes) endet mit RTS => $D078.

$EA4D– RENAME Subdirectories/Files
Der Original-Pathname wird (mit Prefix, falls gesetzt) in den Directory Space ($F600..) kopiert, der neue Path via $EB35 aus dem „user"-Parameterblock geholt und überprüft. Beide Paths werden miteinander verglichen, bis ein unterschiedlicher Name auftaucht. Dieser Name muß in beiden Paths der letzte sein. Sind beide Namen vollständig gleich, erfolgt RTS ohne Fehler. Daß der Diskettenmotor trotzdem kurz läuft, liegt am Suchvorgang für den Original-File.
Als nächstes wird versucht, einen File mit dem neuen Namen auf der Diskette zu finden (FIND FILE ENTRY: $D7AB). Wenn diese Routine nicht mit ERR #46 (FILE NOT FOUND) zurückkehrt, folgt ERR #47: DUPLICATE FILENAME.
Anmerkung: Das ist ein erfreulicher Fortschritt gegenüber DOS 3.3, wo es noch möglich war, mit RENAME mehrere Files mit gleichem Namen zu erzeugen.

Copy & Verify Pathname wird ein weiteres Mal für den alten Pathname benutzt, danach wird jetzt endgültig der Original-File gesucht und die entsprechende Directory geladen (GET FILE: $D798).
Sollte dieser File OPEN sein (existierender FCB-Eintrag), ERR#50: FILE OPEN.
ACCESS (nicht im FCB, sondern im Directory-Eintrag!) wird getestet, falls RENAME nicht erlaubt ist (Bit 6 nicht gesetzt), ERR#4E: FILE ACCESS ERROR.
Sollte der File weder Type 1/2/3, noch Type $0D (Subdirectory) sein, gibt es einen ERR#4A: INCOMPATIBLE FILE FORMAT.
Copy & Verify Pathname wird ein letztes Mal aufgerufen, um den neuen Namen in den MLI-Bereich ($F100-) zu bringen. Den Start des letzten (verschiedenen) Namens innerhalb des neuen Paths hat sich das Programm gemerkt. Mit diesem Teil des neuen Pathname wird jetzt ein Eintrag vorbereitet.
Wenn es sich um eine Subdirectory handelt, muß der neue Name dort ebenfalls eingetragen werden: der Key Block wird gelesen, der Name kopiert und das Ganze zurückgeschrieben.
RENAME (Subdirectories/Files) endet mit ENTRY & UPDATE ($D6BB), wo der File-Eintrag mit dem neuen Namen überschrieben wird. Ein Neusetzen von MODIFICATON DATE & TIME findet dabei nicht statt. RTS von ENTRY & UPDATE aus zu $D078.

**$C3- SET FILE INFO (Start: $E9D4)**

Vorher wurde Copy & Verify Pathname sowie SET TIME aufgerufen.

GET FILE($D798) wird aufgerufen und lädt den File-Eintrag.
Für Volume Directories ist SET FILE INFO nicht erlaubt, GET FILE endet mit ERR#40: INVALID PATHNAME. Dieser Fehler wird von SET FILE INFO nicht abgefangen (Abbruch und Return mit Fehler).
Allerdings kann man bei Subdirectories einen Haufen Unsinn mit diesem Befehl anstellen: Nach Verändern des File Type und/oder Storage Type sind eventuell in der Subdirectory befindliche Files erst einmal nicht mehr zugänglich, bis die Subdirectory wieder korrekt gesetzt wird.
Um mit dem Parameterblock des „user"-Programms einen Directory-Eintrag vorzubereiten (hier: zu ändern), braucht es eine Umsetzungstabelle für die Adressen (s. GET FILE INFO). Hier wird das „Null Field" im Parameterblock übersprungen. Wenn MODIFICATION DATE & TIME spezifiziert ist (<> 0), wird ein Einsprung zu ENTRY & UPDATE benutzt, der hinter der Abfrage der Systemzeit liegt, ansonsten erfolgt der „reguläre" Sprung zu ENTRY & UPDATE ($D6AB). Von da aus RTS => $D078.

**$C4- GET FILE INFO (Start: $E96F)**

Vorher wurde nur Copy & Verify Pathname ausgeführt (vgl. SET F'INFO).

GET FILE($D798) sucht und lädt den File-Eintrag.
Im Gegensatz zu SET FILE INFO ist GET auch für Volume Directories erlaubt. Auf den entsprechenden ERR#40, den GET FILE bei Volume Directories zurückliefert, wird „Space on Volume?" mit Platzbedarf 00 aufgerufen und liefert die Anzahl der freien Blocks auf diesem Volume (VCB, Bytes $14/$15) zurück. Abgezogen von der Gesamtzahl der Blocks (VCB, Bytes $12/$13) ergibt sich die Anzahl der belegten Blocks auf diesem Volume.

Beide Werte (Gesamtzahl und Anzahl der belegten Blocks) werden zwischengespeichert. Die Gesamtzahl der Blocks überschreibt dabei das von GET FILE gesetzte „Aux Info".

$E9B6– Mit Hilfe einer Tabelle zur Adress-Umsetzung werden die durch GET FILE gewonnenen Informationen in den Parameterblock des „user"-Programms kopiert. Diese Tabelle ist dieselbe wie für SET FILE INFO (s. dort), nur daß dieses Mal das „Null Field" ebenfalls beschrieben wird. Es enthält den Storage Type und entweder die Anzahl der belegten Blocks auf diesem Volume oder Nullen.

**$C5– ONLINE (Start: $D42A)**

Vorher ausgeführte Routinen: keine.

ONLINE hat einen je nach Argument verschiedenen Platzbedarf für die zurückgelieferten Daten. Bei spezifizierter Unitnummer sind es $10 Byte, bei Unitnummer = 0 (ALL DEVICES) sind es maximal $100 Bytes.
Abhängig von dieser Bedingung wird die Puffergröße gesetzt. Zusammen mit der im Parameterblock spezifizierten Startadresse ergibt sich der entsprechende Speicherbereich, der dann gegen die System Bit Map geprüft wird ($EE6C). Ist der Speicherbereich belegt, ERR#56: BAD BUFFER ADDRESS.
Der Puffer wird gelöscht. Ist das Argument = 0 (ALL DEVICES), werden alle DEVICEs aus der Global Page kopiert und der Reihe nach (mit dem höchsten Eintrag in der DEVICE TABLE beginnend) via ONLINE SPECIFIC abgearbeitet. Der Index im Puffer wird jeweils entsprechend gesetzt.

$D480– ONLINE SPECIFIC: Im VCB wird ein Eintrag mit dieser Unitnummer oder ein freier Eintrag gesucht ($DA69). Blocknummer 00 02 (Volume Directory) wird gesetzt. Der Block wird gelesen ($DDE1).
Hier wird es logisch etwas komplizierter:
1) Der Eintrag ist neu, d.h. $DA69 hat keinen Eintrag mit dieser Unitnummer gefunden. Die $20 Bytes für den neuen Eintrag werden vollständig gelöscht, es wird überprüft, ob es sich bei dem gelesenen Block um eine ProDOS Volume Directory handelt ($D77D) und der gesamte VCB wird nach einem Eintrag mit dem **Namen** der neuen Directory abgesucht ($DB3D). Existiert ein derartiger Eintrag, muß er sich auf einem anderen Unit befinden, sonst hätte $DA69 ihn bei der Suche nach einem Eintrag via **Unitnummer** bereits entdeckt. Hat dieser Eintrag mit demselben Namen einen offenen File, wird ein Flag ($F075) gesetzt und der neue Eintrag wird nicht gemacht. Ansonsten wird der neue VCB-Eintrag gesetzt (alle diese Routinen: $DAC4).
ONLINE verzweigt an dieser Stelle nur zur Fehler-Routine, wenn der gelesene Block keine ProDOS Volume Directory ist (ERR #52).
Ansonsten wird das Flag $F075 getestet. Wurde ein VCB-Eintrag wegen FILE OPEN auf einem anderen Unit gleichen Namens gemacht, ERR #57: DUPLICATE VOLUME.
Tritt dieser Fehler nicht auf, ist der folgende Vergleich der geladenen Directory mit dem VCB-Eintrag ($DB1C) sinnlos, dieser VCB-Eintrag ist schließlich gerade von der Directory kopiert worden. Er ist nur im Fall 2) von Bedeutung.

2) Ein Eintrag für dieses Unit existiert bereits im VCB. Existiert ein offener File auf diesem Unit, Sprung zum Aufruf von $DB1C (s.o.). In diesem Fall wird getestet, ob sich noch das gleiche Volume in diesem Unit befindet. Ist das nicht der Fall, ERR #2E: DISK SWITCHED AND FILE OPEN.
Existiert kein offener File auf diesem Unit, ist der Ablauf derselbe wie unter 1): der Eintrag wird gelöscht, der geladene Block auf „ProDOS Volume Directory" geprüft, alle anderen VCB-Einträge werden nach dem neuen Namen abgesucht (für DUPLICATE VOLUME), und der Eintrag wird gemacht. Der darauf folgende Vergleich ($DB1C) hat wiederum keine Bedeutung.

Interessanterweise tritt die Fehlermeldung DUPLICATE VOLUME nur auf, wenn auf einem der beiden Volumes gleichen Namens ein File OPEN ist.

Alle Fehler, die während ONLINE SPECIFIC auftreten, werden erst einmal abgefangen, die Fehlernummern landen im Puffer des „user"-Programms ($B4B8).
Tritt kein Fehler auf, wird der Name (+ Unitnummer anstelle des Storage Type) in den Puffer des aufrufenden Programms kopiert.
ONLINE SPECIFIC kehrt zum aufrufenden Programm zurück, d.h. entweder (bei angegebener Unitnummer) zu $D078 oder (Unitnummer = 0, d.h. ALL) zur Schleife in ONLINE selber. Ist die Schleife zu Ende, erfolgt auch hier ein RTS => $D078.

**$C6– SET PREFIX (Start: $D32A)**

Vorher ausgeführte Routinen: keine.

Zuerst wird Copy & Verify Pathname ($D27F) aufgerufen. Kehrt diese Routine mit ERR #40 (INVALID PATHNAME) wieder, wird geprüft, ob dieser Fehler durch eine Namenslänge von Null (d.h. „/") hervorgerufen wurde. In diesem Fall wird PREFIX ($BF94) auf Null gesetzt, d.h. CLEAR PREFIX, gefolgt von RTS => $D078. Andere Fehler in Copy & Verify Pathname sind nicht zulässig.
$D339– GET FILE ($D798) wird aufgerufen und lädt den File-Eintrag. Wenn diese Routine mit ERR#40 zurückkehrt, ist das ebenfalls kein Fehler: wir befinden uns dann in einer Volume Directory, d.h. der Path besteht nur aus einem Namen. GET FILE hat in diesem Fall den Storage Type auf „Subdirectory" getürkt.
Der gefundene Storage Type wird überprüft. Es muß sich um eine Subdirectory handeln, ansonsten ERR#4B: UNSUPPORTED STORAGE TYPE.
$D3AB– Die Länge des Pathname wird überprüft. Je nachdem, ob ein vollständiger oder unvollständiger („partial") Pathname spezifiziert wurde, geht das frühere Prefix mit in die Rechnung ein. Überschreitet die Gesamtlänge den Wert $40, ERR#40: INVALID PATHNAME.
Die Blocknummer und die Unitnummer des gefundenen Volumes werden gespeichert (**nicht** im VCB!).
$D372– Der Prozeß des Anhängens an bzw. Überschreibens des alten Prefixes ist raffiniert programmiert worden und in „Programmiertricks" erklärt. SET PREFIX endet mit RTS => $D078.

**$C7– GET PREFIX (Start: $D382)**

Vorher ausgeführte Routinen: keine.

Die Puffer-Startadresse wird aus dem „user"-Parameterblock geholt. Der benötigte Speicherplatz beträgt $40 Bytes. Der dadurch belegte Speicherraum wird mit der System Bit Map gegengeprüft ($EE6C). Ist dieser Platz belegt, folgt ERR#56: BAD BUFFER ADDRESS. Das momentane Prefix wird in den Puffer kopiert. Dabei werden die Längen-Bytes der einzelnen Namen wieder durch Schrägstriche ersetzt (s. Beschreibung von Copy & Verify Pathname in „Programmiertricks"). Am Ende des Pathname wird ein zusätzlicher Schrägstrich angehängt. RTS => $D078

**$C8– OPEN (Start: $E0B9)**

Vorher wurden Copy & Verify Pathname sowie SET TIME ausgeführt.

GET FILE ($D798) sucht und lädt den File-Eintrag.
Da OPEN auch für Volume Directories zulässig ist, wird in diesem Fall die entsprechende Fehlermeldung von GET FILE (ERR #40) maskiert.
Der FCB wird nach einem File dieses Namens abgesucht ($E1A9). Existiert ein FCB-Eintrag für diesen File, ERR #50: FILE (ALREADY) OPEN.
Gleichzeitig hat $E1A9 nach einem freien Eintrag im FCB gesucht. Falls kein Eintrag gefunden wurde, ERR #42: MAXIMUM NUMBER OF FILES OPEN.
Ansonsten wird der freie FCB-Eintrag vollständig gelöscht, Unitnummer, Blocknummer des (Sub)directory Key Blocks, Nummer des Directoryblocks, in dem der Eintrag für diesen File steht und Eintragsnummer in diesem Block werden in den FCB-Eintrag kopiert.
$E0FE– Für Datenfiles werden die R/W-Bits von ACCESS direkt in Byte $09 des FCB-Eintrags kopiert, für (Sub)directories ist nur READ erlaubt.
Die „MLI Version" des Files wird überprüft.
$E123– Überprüfung des Storage Type. OPEN läßt nur die Typen 1/2/3 und $0D (Subdirectory oder von GET FILE „getürkte" Volume Directory) zu.
$E12B– Die Nummer des „ersten Blocks im File", die Anzahl der vom File insgesamt belegten Blocks und der EOF werden kopiert. Dazu ist wieder die bereits bei GET/SET FILE INFO verwendete Adressumsetzungstabelle notwendig.
$E144– Erst jetzt wird der FCB-Eintrag durch Setzen der File Reference Number (Byte $00) gültig gemacht. Die vom „user"-Programm spezifizierte Pufferadresse wird vom Parameterblock geholt, der Pufferbereich gegen die System Bit Map geprüft, belegt und in der Global Page eingetragen ($EDD7). Ein Teil von „Get File Data" ($D3E0) wird benutzt, um die Pufferpointer auf der Zeropage zu setzen und den zum Unit gehörigen VCB-Eintrag zu finden.
$E15F– Für Datenfiles: Die „Position jetzt im File" wird auf einen unmöglichen Wert ($FF 00 00) gesetzt, die „Position gewünscht" auf 00 00 00. SET MARK wird aufgerufen und liest den ersten Datenblock in den Filepuffer (s. WRITE im vorigen Kapitel zu SET MARK).
Wenn SET MARK ohne Fehler zurückkehrt, wird die Anzahl der offenen Files für dieses Unit im VCB-Eintrag erhöht und „File(s) OPEN" (VCB-Byte $11) gesetzt. RTS => $D078
$E173– Bei einem Fehler innerhalb von SET MARK wird der dem File zugeordnete Puffer

wieder in der System Bit Map freigegeben und der FCB-Eintrag gelöscht. Danach erfolgt RTS zu $D078 mit gesetzter ERR#.
$E18A– Für Directory-Files: Die Blocknummer des Key Blocks wurde bei $E12B gesetzt, der Block wird gelesen ($DFF7) und die Blocknummer als „Datenblock jetzt im Puffer" (FCB- Bytes $10/$11) verzeichnet. Passiert dabei ein Fehler, Sprung zu $E173. Ansonsten RTS => $D078.

**$C9– NEWLINE (Start: $E95D)**

Vorher wurde Set File Data aufgerufen.

Set File Data hat den entsprechenden FCB-(Basis)index gesetzt. Die beiden Bytes (ENABLE MASK und NEWLINE CHARACTER) werden aus dem Parameterblock des „user"-Programms geholt und im FCB-Eintrag gespeichert. RTS => $D078

**$CA– READ (Start: $E1F7)**

Vorher wurde Set File Data aufgerufen.

Die Startadresse des Zielbereichs (**nicht des Filepuffers!**) wird aus dem Parameterblock des „user"-Programms geholt und auf die Zeropage transportiert ($E403).
Der „Request Count" wird aus dem Parameterblock geholt und zweifach (Pufferlänge/Transferzähler) gespeichert ($E3E8).
Die „Position jetzt im File" wird kopiert, der Request Count aufaddiert und das Ergebnis mit dem EOF des Files verglichen ($E415).
$E201– Wenn READ nicht erlaubt ist (FCB-Byte $09, Bit 0 nicht gesetzt), ERR#4E: FILE ACCESS ERROR.
$E20A– War „Position jetzt" + Request Count > EOF, wird die Anzahl der zu transferierenden Bytes herabgesetzt. Der Transport endet dann immer auf dem EOF. Bleibt dabei nichts mehr übrig ($0 Bytes zu transferieren), ERR #4C: EOF ENCOUNTERED.
$E23C– Der Zielbereich wird gegen die System Bit Map geprüft ($EE6C).
$E241– Der Storage Type des Files wird ermittelt ($E40E). Für Directory-Files verläuft der Transfer anders. Sprung zu $E3B1.
$E24B– Mit SET MARK wird der entsprechende Block in den Filepuffer geladen. Der Transport des ersten Blocks in den Zielbereich erfolgt in jedem Fall Byte-per-Byte ($E330).
$E256– Wenn der Request Count abgearbeitet oder der Transfer durch NEWLINE beendet wurde, kehrt READ via $E2EF („Set Transfer Count") zurück.
$E258– Wenn NEWLINE aktiv ist, geht der Transfer auf Byte-per-Byte Basis weiter, desgleichen, wenn nur noch weniger als ein Block ($200 Byte) zu lesen ist. Eine letzte Möglichkeit gibt es noch, READ davon abzuhalten, den „zweiten Gang" einzulegen: wenn dieser erste Datenblock vorher (im Speicher) mit WRITE verändert wurde, muß SET MARK ihn als nächstes zurückschreiben (s. nächster Absatz).
$E26D– Der „zweite Gang" wird einlegt – SET MARK bekommt eine neue Pointeradresse und liest beim nächsten Aufruf nicht mehr in den Filepuffer, sondern direkt in den Zielbereich. Ab jetzt müssen Fehler abgefangen werden, weil der Pointer zum Filepuffer nicht mehr stimmt. Das eventuell notwendige Zurückschreiben eines geänderten Datenblocks ist damit auch nicht mehr möglich (s.o.).

$E295$- Bis jetzt sind maximal $3FF Bytes gelesen worden. Bleibt kein ganzer Block mehr übrig, werden die Filepuffer-Pointer wieder zurückgesetzt ($E3A3). Bleibt überhaupt nichts mehr übrig, RTS via „Set Transfer Count" ($E2EF). Ansonsten werden die letzten Bytes wieder Byte-per-Byte transportiert.
$E2A7– Wenn sich die „Position jetzt im File" auf einer Index-Blockgrenze befindet, geht es weiter mit dem Block-Transport via SET MARK.
$E2A9– Jetzt kommt READ erst richtig auf Touren. Aus dem Indexblock wird die nächste Blocknummer gelesen und die entsprechende RWTB direkt (via Unitnummer) aufgerufen. Das Laden von 8 kByte (ein Hires-Bild) dauert ab hier noch knapp zwei Sekunden!
Wird während dieser Schleife ein „Loch" (Blocknummer 00 00, d.h. „sparse Block") gefunden oder bleibt weniger als ein Block übrig, übernimmt die Schleife mit SET MARK wieder das Kommando. Für „sparse Blocks" löscht SET MARK das Zielgebiet, für einzelne Bytes wird wieder auf den Byte-per-Byte Transfer verzweigt.
Am Ende des Loops von „Stufe 3" werden die Pointer zum Filepuffer wieder ordentlich gesetzt ($E3A3) und READ endet via „Set Transfer Count" ($D2EF). Diese Routine endet mit SET MARK. Wurde der letzte Transfer Byte-per-Byte ausgeführt, dann stimmt die „Position jetzt" im FCB-Eintrag nicht mit der „Position gewünscht" überein bzw. ist mindestens einen ganzen Block „weiter". SET MARK liest dann den nächsten Block wieder in den Filepuffer ein oder löscht den Filepuffer, wenn der EOF des Files erreicht ist. RTS von SET MARK zu $D078.
$E3B1– READ für Directory-Files
Diese Routine enthält eine Hauptschleife, die zuerst SET MARK aufruft und dann die Kopier-Routine von READ ($E330) benutzt.
Es gibt drei Möglichkeiten, die Schleife zu beenden: NEWLINE, Transfer-Zähler auf 00 heruntergezählt oder einen Fehler in SET MARK.
Wird der Loop fehlerfrei beendet, wird „Set Transfer Count" aufgerufen. Ergibt sich beim letzten SET MARK der ERR#4C: EOF ENCOUNTERED, wird der Datenteil des Filepuffers gelöscht („normal") und die Blocknummer in den Bytes $00 und $01 des Datenteils eingetragen (s. „Spezielles"). Danach RTS zu $D078 mit Carry gelöscht, d.h. ohne Fehler.
Wird der Loop aufgrund eines Fehlers verlassen, wird vor dem Abbruch von READ noch der Transfer Count gesetzt ($E2E8). In allen Fällen: RTS => $D078.

**$CB–WRITE (Start: $E47C)**

Vorher wurde Set File Data aufgerufen.

Die Struktur von WRITE ist bereits recht ausführlich im vorigen Kapitel besprochen worden, deshalb nur noch einmal das Wichtigste:
Alle hier auftretenden Fehler verzweigen zu einer Routine, die die neue „Position im File" sowie den neuen EOF im FCB speichert ($E442), danach „Set Transfer Count" ($E27F) durchführt und erst dann mit der Fehlernummer abbricht.

$E47C– Der Request Count wird aus dem Parameterblock des „user"-Programms geholt ($E3E8).
Die „Position jetzt im File" wird aus dem FCB-Eintrag geholt, der Request Count addiert und das Ergebnis gegen den EOF des Files geprüft ($E415). Im Gegensatz zu READ wird hier

der EOF im FCB-Eintrag neu gesetzt (falls überschritten). SET MARK bezieht sich aber immer auf eine vorher erstellte Kopie des alten EOF.
Wenn WRITE nicht zulässig ist (FCB-Byte $09, Bit 1 nicht gesetzt), ERR#4E: FILE ACCESS ERROR.
$E494– Wenn der Request Count > 0 ist, wird der Start des Zielpuffers aus dem Parameterblock des „user"-Programms geholt und auf die Zeropage kopiert ($E403).
$E4A4– Falls der Storage Type nicht 1/2/3 ist (WRITE ist nur für Datenfiles!), ERR#4E.
$E4A6– SET MARK wird aufgerufen und bringt entsprechend gesetzte Erweiterungsbits in STATUS (FCB-Byte $08, Bit 0-2) zurück, falls eine Erweiterung des Storage Type oder ein neuer Datenblock benötigt wird.
$E4B0– Falls keine neuen Blocks oder Erweiterungen benötigt werden, Sprung zu $E514. Ansonsten wird über die Anzahl der gesetzten Bits im Bereich Bit 0-2 von STATUS die Anzahl der benötigten Blocks festgestellt und „Space on Volume?" ($DB66) aufgerufen.
Für Erweiterung auf Type 3 wird $E58F aufgerufen:
ALLOCATE NEXT BLOCK wird aufgerufen und belegt den zukünftigen Masterblock. Die Nummer des momentan einzigen Indexblocks wird in den Indexteil des Filepuffers kopiert (der von SET MARK aufgrund „neuer Indexblock benötigt" vorher gelöscht wurde). Der Nummer des „ersten Blocks im File" wird auf den Masterblock gesetzt, die alte Nummer des „ersten Blocks" (= bis jetzt einziger Indexblock) landet als erste Blocknummer im Masterblock. Weiter geht es mit derselben Routine, die für das Anhängen eines weiteren Indexblocks an einen File mit dem Storage Type 3 benutzt wird: hier wird via ALLOCATE NEXT BLOCK ein Indexblock belegt und die Blocknummer in den Masterblock eingetragen. Der Indexteil des Filepuffers (= Masterblock) wird zurückgeschrieben. Danach wird der Indexteil gelöscht, hier wird dann die Nummer des nächsten Datenblocks eingetragen.
Für Erweiterung auf Type 2 wird $E5DA aufgerufen:
ALLOCATE NEXT BLOCK belegt den zukünftigen Indexblock auf dem Volume. Die Nummer des „ersten Blocks im File" wird auf den gerade belegten Block gesetzt, die alte Nummer des „ersten Blocks" landet als erster Eintrag im Indexteil des Filepuffers, der vorher von SET MARK gelöscht wurde („neuer Indexblock benötigt").
$E4E3– Die Erweiterung des Storage Type hat stattgefunden. Der nächste freie Datenblock wird gesucht und belegt (ALLOCATE NEXT BLOCK, $E62E). Bit 7 in STATUS wird gesetzt („geänderter Indexblock im Speicher"). Anmerkung: Wenn ein weiterer Block im File belegt wird, haben wir in jedem Fall mindestens Storage Type 2, der erste Datenblock wurde bereits von CREATE belegt.
Über die „Position jetzt im File" wird die Eintragsposition der neuen Nummer des neuen Blocks im Indexblock bestimmt und der Eintrag gemacht. „Datenblock momentan im Speicher" (FCB-Bytes $10/$11) werden ebenfalls auf diese Blocknummer gesetzt. SET MARK hat den Datenteil des Filepuffers gelöscht („neuer Datenblock benötigt"), d.h. der neue Datenblock ist (im Speicher) leer.
$E514– Der Datenblock, auf den geschrieben werden soll, befindet sich (jetzt) im Speicher, die zu schreibenden Daten werden vom Puffer des „user"-Programms Byte-per-Byte kopiert ($E51F). NEWLINE ist zwar kopiert worden (falls gesetzt), wird aber nicht berücksichtigt. Im Gegensatz zu READ gibt es bei WRITE keinen „Rallygang" (auch wenn das theoretisch möglich wäre), weil vor jedem Block gegen EOF getestet werden muß und gegebenenfalls eine Erweiterung des Files bzw. des Storage Type stattfindet. Deshalb findet nach Ende der Kopierroutine ein Sprung zurück nach $E4A6 statt, falls noch Daten zu schreiben sind.
Ist alles geschrieben, endet WRITE über „Set Transfer Count" ($E2EF), wobei SET MARK

noch einmal aufgerufen wird. Der letzte beschriebene Datenblock wird dabei nur dann sofort auf das Volume geschrieben, wenn sich die „Position jetzt im File" genau auf dem Anfang des nächsten Datenblocks befindet. RTS von SET MARK aus zu $D078.

**$CC– CLOSE (Start: $E677)**

Vorher wurde nur SET TIME durchgeführt.

Alle Fehler, die während der Ausführung auftreten, führen nicht zu einem Abbruch, sondern verzweigen zu einer Fehlerbehandlungsroutine ($E7ED), wo geprüft wird, ob es sich um CLOSE SPECIFIC oder CLOSE ALL handelt. Bei CLOSE SPECIFIC erfolgt Abbruch, bei CLOSE ALL wird die Fehlernummer zwischengespeichert und der „ALL"-Loop fortgesetzt, bis alle Files CLOSEd sind.

$E677– Die File Reference Number wird aus dem Parameterblock des „user"-Programms geholt. Hier ist auch der Wert $00 (= ALL) als Argument möglich. Falls eine spezifische Reference Number angegeben wurde, verzweigt die Ausführung zu $E6B2.
$E682– CLOSE ALL: In einer Schleife wird der gesamte FCB abgesucht. Falls ein besetzter Eintrag (= offener File) gefunden wird, erfolgt ein FLUSH ($E714) dieses Files sowie ein Aufruf von CLOSE SPECIFIC ($E6B7, zweiter Einsprung). Ist der letzte Eintrag bearbeitet, wird die eventuell zwischendurch gesetzte Fehlernummer geladen. RTS => $D078.
$E6B2– CLOSE SPECIFIC: FLUSH SPECIFIC ($E71C) wird aufgerufen. Über den Pufferindex (FCB-Byte $0B) wird der Filepuffer bestimmt und in der System Bit Map wieder freigegeben ($EE34).
Der FCB-Eintrag wird durch Löschen der Reference Number (Byte $00) und der Unitnummer (Byte $01) ungültig gemacht.
Der VCB-Eintrag für dieses Unit wird gesucht ($DA69) und die Anzahl der offenen Files auf dem Unit um eins verringert (VCB-Byte $1E). Bleiben dabei keine offenen Files auf diesem Unit mehr übrig, wird das Flag „OPEN File(s) on Unit" (VCB-Byte $11) zurückgesetzt.
CLOSE SPECIFIC kehrt je nach Aufruf entweder zu CLOSE ALL oder zu $D078 zurück.

**$CD–FLUSH (Start: $E6E8)**

Vorher wurde nur SET TIME ausgeführt.

Alle während FLUSH auftretenden Fehler (Ausnahme: SYSTEM DEATH) verzweigen zu derselben Routine, die von CLOSE im Fehlerfall benutzt wird ($E7ED).
$E6E8– Die File Reference Number wird aus dem Parameterblock des „user"-Programms geholt. Ist die Reference Number > 0, erfolgt ein Sprung zu FLUSH SPECIFIC ($E71C).
$E6F3– Der gesamte FCB wird in einer Schleife abgesucht. Wird dabei ein Eintrag (= offener File) gefunden, erfolgt FLUSH SPECIFIC ($E714, zweiter Einsprung). Ist der letzte FCB-Eintrag behandelt, wird die eventuell inzwischen gesetzte Fehlernummer geladen. RTS => $D078.
$E714– FLUSH SPECIFIC, zweiter Einsprung: ein Teil von Set File Data ($D3E0) wird benutzt, um die Adresse des Filepuffers und die entsprechenden Pointer auf der Zeropage zu setzen. Die Programmausführung wird auf $E726 fortgesetzt.

$E71C– FLUSH SPECIFIC, Einsprung für angegebene Reference Number:
Set File Data wird (vollständig, Überprüfung der Reference Number) aufgerufen.
$E726– Files, bei denen WRITE nicht erlaubt ist (FCB-Byte $09, Bit 1 nicht gesetzt), brauchen nicht behandelt zu werden (CLC/ RTS).
$E72D– Wenn während WRITE der EOF des Files verändert wurde (FCB-Byte $1C, Bit 7 gesetzt), wird FLUSH immer ausgeführt. Ansonsten wird der STATUS des Files getestet: befindet sich kein geänderter Datenblock (Bit 6) im Speicher oder wurde der File überhaupt nicht verändert ($Bit 4), ist die Routine hier ebenfalls zu Ende.
$E739– Abhängig von STATUS werden Index- und/oder Datenblock auf die Diskette zurückgeschrieben.
$E751– Ein Directory-Eintrag (Update) wird vorbereitet: Bytes $01-$06 des FCB-Eintrags werden kopiert, der Key Block der Directory, in dem dieser File verzeichnet ist, wird gelesen. Die ensprechenden Parameter dieses Key Blocks (Parent Pointer etc.) werden kopiert ($D8A0). Wenn der File-Eintrag nicht im Key Block selber steht, wird danach der entsprechende Directoryblock gelesen.
Der File-Eintrag wird lokalisiert ($D692) und herauskopiert ($D79D).
$E794– Die Anzahl der von diesem File belegten Blocks, der EOF, die Nummer des „ersten Blocks im File" werden im herauskopierten File Info überschrieben, der Storage Type wird auf den neuesten Stand gebracht. Damit haben wir jetzt einen vollständigen Directory-Eintrag (im selben Format wie für einen neuen File). ENTRY & UPDATE ($D6AB) wird aufgerufen, überschreibt den alten File-Eintrag in der Directory und arbeitet sich von da aus bis zur Volume Directory hoch.
$E7D6– Das Flag „File altered" (FCB-Byte $1C) wird zurückgesetzt. Die eventuell gesetzten Bits in STATUS sind bereits gelöscht, d.h. ein erneutes (direkt folgendes) FLUSH für diesen File kehrt sofort zurück. Falls die VBM dieses Volumes noch (verändert) im Speicher ist (d.h. noch kein anderer File dieses Volumes geFLUSHt wurde), wird sie zurückgeschrieben ($DD86). RTS entweder zur Schleife von FLUSH ALL, zum aufrufenden CLOSE oder zu $D078.

**$CE– SET MARK (Start: $DE1E)**

Vorher wurde Set File Data aufgerufen.

Die gewünschte Position wird aus dem Parameterblock des „user"-Programms geholt und mit dem momentanen EOF des Files (FCB-Bytes $13-$15) verglichen. Ist die gewünschte Position > EOF, erfolgt Abbruch mit ERR#4D: POSITION OUT OF RANGE.

$DE3B– SET MARK: Einsprung für alle internen Aufrufe. Diese Funktion ist auch bereits im vorigen Kapitel recht ausführlich (zusammen mit WRITE) behandelt worden.
Der Terminus „neuer Indexblock" steht für „Indexblock momentan nicht im Speicher" und kann deshalb auch für Files des Storage Type 2 zutreffen, weil OPEN und SET EOF diese Routine ebenfalls benutzen. In diesem Fall wird eine „Position jetzt" gesetzt, die die maximale Größe eines Files vom Typ 3 überschreitet (OPEN) bzw. die „Position jetzt" gründlich verdreht (SET EOF). Folge: SET MARK lädt aufgrund der Positionsdifferenz (gewünschte Position von OPEN: 00 00 00, von SET EOF: vorherige „Position jetzt") für Type 3 zuerst den Masterblock und für Type 2 den Indexblock.

Für Type 2 ist der „neue Indexblock" dabei nach wie vor der einzige existierende Indexblock. Er befindet sich aber beim Aufruf via OPEN noch nicht im Filepuffer. Bei SET EOF wurde in diesem Indexblock eventuell gelöscht – er befindet sich aber nicht im Filepuffer, weil SET EOF die Routine REMOVE BLOCKS gemeinsam mit DESTROY benutzt, wobei bei DESTROY kein Filepuffer zugeordnet sein kann (File nicht OPEN). REMOVE BLOCKS arbeitet deshalb im Directory Space ($F600/$F700) und schreibt von da aus direkt auf das Volume zurück, also bei offenen Files am Filepuffer „vorbei".
Die Differenz zwischen „Position jetzt" und „Position gewünscht" wird berechnet. Ergibt sich eine Differenz im Bereich von $00 00 bis + $01 FF, ist der gewünschte Block bereits im Speicher.
Ansonsten wird über den Absolutbetrag der „Position gewünscht" festgestellt, ob der momentane Storage Type überschritten wurde. Ist das der Fall, werden die entsprechenden Erweiterungsbits in STATUS gesetzt und die entsprechenden Teile des Filepuffers (Daten/Index) gelöscht. Diese Löschung erfolgt auch für „sparse"-Blocks oder das Anhängen eines weiteren Indexblocks an einen File vom Type 3 bzw. das Anhängen eines Datenblocks an einen File vom Type 2 oder 3 – hier findet dann natürlich keine Veränderung des Storage Type statt. In die gelöschten Teile des Filepuffers kann dann WRITE sofort hineinschreiben. Das Belegen der neuen Blocks auf dem Volume erfolgt in WRITE selber!
Für den Fall, daß der entsprechende Block bereits im File existiert, wird er (ggf. durch Lesen des entsprechenden Indexblocks) bestimmt und in den Filepuffer (Datenteil) gelesen. Muß zum Lesen eines Indexblocks zuerst der Masterblock des Files (Type 3) gelesen werden, wird hierzu der Datenteil des Filepuffers benutzt, das anschließende Lesen des Datenblocks bzw. Löschen des Datenteils überschreibt dann die Daten des Masterblocks wieder im Filepuffer. In beiden Fällen endet SET MARK mit „BLOCK IN MEMORY", die neue „Position jetzt" wird im FCB-Eintrag gesetzt. RTS zur aufrufenden Routine oder zu $D078.

$DFAE– SET MARK FÜR DIRECTORY FILES
Ist der File keine (Sub)directory, ERR#4A: INCOMPATIBLE FILE FORMAT. Exit via SET SYSERR zur aufrufenden Routine oder zu $D078.
Ansonsten wird anhand der Differenz „Position jetzt" - „Position gewünscht" bestimmt, ob die Suche über die „Forward Reference" oder die „Backward Reference" der Directory-Blocks laufen soll. Ergibt sich bei dieser Differenzbestimmung ein Wert im Bereich von $00 00 bis + $01 FF, Sprung zu BLOCK IN MEMORY ($DF7C).
Die Differenz (in ganzen Blocks) wird als Zähler gespeichert.
$DFC6– Die Suche geht vorwärts: Der nächste Directoryblock wird via Forward Reference des zuletzt geladenen Blocks bestimmt und geladen ($DFE2). Der Zähler wird heruntergezählt. Am Schleifenende Sprung zu BLOCK IN MEMORY.
$DFD4– Suche rückwärts: Der nächste Directoryblock wird via Backward Reference bestimmt und geladen ($DFE2). Der (negative) Zähler wird heraufgezählt. Am Schleifenende Sprung zu BLOCK IN MEMORY.
$DFE2– Die Reference wird aus dem zuletzt geladenen Directoryblock gelesen. Ergibt sich dabei die Blocknummer 00 00, ist die Directory zu Ende: ERR #4C: EOF ENCOUNTERED. Ansonsten wird der durch die Reference bestimmte Block in den Datenbereich des Filepuffers gelesen und „Datenblock jetzt im Speicher" im FCB-Eintrag gesetzt ($DFF7). RTS zum laufenden Loop ($DFC6 oder $DFD4).

**$CF– GET MARK (Start: $DE08)**

Vorher wurde Set File Data aufgerufen.

Die „Position jetzt im File" (FCB-Bytes $12-$15) wird in den Parameterblock des „user"-Programms kopiert, der FCB-(Basis)index ist durch Set File Data gesetzt worden. RTS zu $D078.

**$D0– SET EOF (Start: $E807)**

Vorher wurde Set File Data ausgeführt.

$E807– Der Storage Type des Files wird aus dem FCB geholt. Das Setzen des EOF in Directory Files ist nicht erlaubt: ERR #4E: FILE ACCESS ERROR. SET EOF in Files, bei denen WRITE nicht erlaubt ist (FCB-Byte $09, Bit 1 nicht gesetzt) führt zu demselben Fehler.
$E821– Der momentane EOF wird aus dem FCB-Eintrag geholt und mit dem gewünschten EOF aus dem Parameterblock des „user"-Programms verglichen.
$E84E– Der gewünschte EOF ist größer als der alte EOF: Der alte EOF im FCB-Eintrag wird mit dem neuen EOF aus dem Parameterblock überschrieben. Byte $1C im FCB-Eintrag (EOF geändert) wird gesetzt ($EC50). RTS => $D078.
$E863– Der gewünschte EOF ist kleiner als der alte EOF: FLUSH wird aufgerufen, um eventuell geänderte Daten/Indexblocks des Files auf das Volume zurückzuschreiben.
$E872– Die „Position jetzt im File" wird mit dem gewünschten EOF verglichen. Steht sie hinter dem gewünschten EOF, wird sie auf den EOF zurückgesetzt.
$E898– Wenn der neue EOF genau auf einer Blockgrenze endet, wird der zusätzliche Block nicht benötigt und kann ebenfalls gelöscht werden. (Beispiel: EOF = $400, => 2 Datenblocks; EOF $401 => 3 Datenblocks). Falls diese Herabsetzung eine negative Zahl ergibt, wird die Untergrenze für die Löschung auf 00 00 00 korrigiert.
$E8D8– Die (momentane) Nummer des „ersten Blocks im File" – sie kann sich durch Herabsetzung des Storage Type ändern – wird aus dem FCB-Eintrag kopiert. Die Anzahl der bis jetzt freigegebenen Blocks wird auf Null gesetzt und REMOVE BLOCKS ($EC62) wird aufgerufen. REMOVE BLOCKS gibt die entsprechenden Blocks wieder frei und setzt ggf. den Storage Type des Files herab. Dieses Herabsetzen des Storage Type ist eine exakte Umkehrung des Erweiterungsprozesses in WRITE (s. dort) – mit einem Unterschied: da REMOVE BLOCKS auch von DESTROY benutzt wird, findet die Bearbeitung der Blocks (Lesen und Löschen von Blocknummern in Master/Indexblocks) im Directory Space ($F600/$F700) statt. Bearbeitete Master/Indexblocks werden von da aus auf das Volume zurückgeschieben und müssen via SET MARK neu in den Filepuffer geladen werden.
$E8F5– Die Nummer des „ersten Blocks im File" wird in den FCB-Eintrag zurückkopiert (s.o.), die Anzahl der vom File belegten Blocks (FCB-Bytes $18/$19) wird auf den neuesten Stand gebracht, genauso wie der Storage Type.
Die Anzahl der freien Blocks auf dem Volume wird entsprechend erhöht ($EBDD).
$E928– Die „Position jetzt im File" wird als „Position gewünscht" kopiert und der FCB-Eintrag „Position jetzt" gründlich verbogen. Im darauf folgenden Aufruf „glaubt" SET MARK, daß der entsprechende Block nicht im Speicher ist und lädt den Datenblock (+ evtl.

Indexblock) neu in den Filepuffer. Das ist notwendig: es kann sein, daß REMOVE BLOCKS im „momentanen" Daten/Indexblock Daten gelöscht hat.
Schlußendlich wird der neue EOF auch im FCB-Eintrag gesetzt ($E84E), eventuell bei SET MARK gesetzte Fehlernummern werden geladen. RTS => $D078.

**$D1– GET EOF (Start: $E94B)**

Vorher wurde Set File Data ausgeführt.

Der momentane EOF des Files (FCB-Bytes $15-$17) wird in den Parameterblock des „user"-Programms kopiert, der FCB-(Basis)index ist durch Set File Data gesetzt worden. RTS => $D078.

**$D2– SET BUF (Start: $EEA7)**

Vorher wurde Set File Data ausgeführt.

Die zum File gehörige Puffer-Startadresse wurde dabei via Pufferindex (FCB-Byte $0B) aus der Tabelle in der Global Page kopiert.
Die neu spezifizierte Filepuffer-Startadresse wird aus dem Parameterblock des „user"-Programms geholt und der Pufferbereich gegen die System Bit Map geprüft ($EDD9).
Wenn der Bereich frei ist, wird der alte Filepuffer in der System Bit Map freigegeben ($EE43).
Der Inhalt des alten Puffers wird vollständig in den neuen Puffer kopiert ($400 Byte). RTS => $D078.

**$D3– GET BUF (Start: $EE98)**

Vorher wurde Set File Data ausgeführt (s. SET BUF).

Die Startadresse des Filepuffers wird in den Parameterblock des „user"-Programms kopiert. RTS => $D078.

# 1.5. Das MLI in Blöcken

Anmerkung: In dieser Beschreibung wird von einem APPLE mit 64k/128k RAM ausgegangen. Für 48k verschieben sich alle Adressen um $3000 nach unten.

Alle MLI-Befehle (Ausnahme: REBOOT) springen nach der Ausführung via $D078 zu MLIEXIT in der Global Page und kehren von da aus zum aufrufenden Programm („user") zurück. Das MLI hat einen einzigen Startpunkt, die Adresse $D000. (In Version 1.1.1 geändert!)

$D000-$D077 **Einsprung in das MLI mit JMP von der Global Page.** X- und Y-Register werden auf der Global Page gespeichert, die Returnadresse zum „user"-Programm wird vom Stack geholt, um vier erhöht und in MLIRTS gespeichert.
Aus der urspünglichen Returnadresse ergibt sich die Adresse des Command-Parameters, die als Pointer auf der Zeropage gespeichert wird. SYSERR wird auf $00 gesetzt.

$D01D Das Command-Byte wird geladen und auf den Bereich $00-$1F umgerechnet. Die Reihenfolge der Kommandos bleibt dabei erhalten (Ausnahme: $D3 wird zu $00).

$D02C Ist das Command nicht in der Tabelle auf $EF25 enthalten, ERR #1: BAD SYSTEM CALL NUMBER.

$D031 Die Startadresse des Parameterblocks wird als Pointer auf der Zeropage gespeichert.

$D03F Die Anzahl der Parameter (Byte $00 des Parameterblocks) wird mit einer Tabelle auf $EF45 verglichen. Das einzige Command mit 0 Parametern ist $82 (SET TIME) => Sprung zu $D060. Stimmt die Anzahl der Parameter nicht überein, ERR #4: BAD SYSTEM CALL PARAMETER COUNT.

$D04B Test auf $65 (REBOOT). REBOOT hat erstaunlicherweise einen obligatorischen Parametersatz, der allerdings nie gebraucht wird („dummy"). => Sprung zu $D05D.

$D04F „Verteilung" der Befehle auf drei Klassen: $4x/$8x/$Cx-$Dx.

$D057 Für $40/$41: Aufruf von $D0F3, Sprung zu $D078.

$D060 REBOOT: Sprung zu $BF03. Kein Return!

$D063 SET TIME: Aufruf von $BF06, Sprung zu $D078.

$D066 Für $80/$81: Umwandlung auf $01/$02, Aufruf von $D0B2, Sprung zu $D078.

$D071 Für $Cx-$Dx: Weitere Umrechnung des Index (vom Bereich $00-$1F auf $00-$13), Aufruf von $D23C. Danach weiter mit $D078.

$D078 **EXIT**
Das „Backup"-Flag auf der Global Page wird gelöscht, durch Vergleich von SYSERR mit $01 wird das Carry gesetzt, falls SYSERR nicht während der Befehlsausführung Null geblieben ist. MLIAKTIV wird auf „nicht aktiv" rotiert, die Returnadresse zum „user"-Programm von MLIRTS geholt und gepusht, ebenso wie das durch den Vergleich mit SYSERR entsprechend gesetzte Carry. X- und Y-Register werden von der Global Page zurückgeladen, das ID der vor dem MLI-Aufruf aktiv gewesenen Bank wird geladen. Sprung zu $BFA0 (MLIEXIT).

$D0A2 **NO DEVICE**
Alle unbesetzten RWTB-Startadressen in der Global Page zeigen hierher. Setzt SYSERR #28: NO DEVICE CONNECTED. RTS mit Carry gesetzt via $BF09 (immer zur Routine, die $D0DA aufgerufen hat).

$D0A7 Setzt (SYS)ERR#1: BAD SYSTEM CALL NUMBER. Sprung zu $D078.

$D0AB Setzt (SYS)ERR#4: BAD SYSTEM CALL PARAMETER COUNT. Sprung zu $D078.

$D0B2-$D0D9 **Handler für $80(READ) und $81(WRITE).**

$D0B4 Abschalten der Interrupts.
Befehl, Unitnummer, Ziel/Quelladresse und Blocknummer werden für die RWTB aus dem Parameterblock des „user"-Programms auf die Zeropage kopiert.
Ein Block belegt normalerweise 2 Speicherseiten. Ist die Startadresse nicht auf einer Seitengrenze, werden drei Speicherseiten als belegt angenommen. Test, ob der Bereich in der SBM frei ist ($EE89). Dieser Test erfolgt für READ und(!) WRITE.
Überschreitet die Obergrenze des Puffers $BFFF, ist das Ergebnis des Vergleichs mit der SBM „random" (s. „Spezielles").

$D0CE Aufruf der RWTB des entsprechenden Units via $D0DA.

$D0D3 Kein Fehler: Wiederherstellung des INT-Status und RTS => $D078.

$D0D6 Fehler: Wiederherstellung des INT-Status und RTS (=> $D078) via SET SYSERR ($BF09).

$D0DA-$D0F2 **DO DISK I/O**
Jeder Disk I/O des MLI läuft über diese Routine.
Über die Unitnummer wird die Startadresse der RWTB aus der Global Page geholt und dorthin gesprungen. Die Unterscheidung auf „aktivate", READ, WRITE und FORMAT muß in der RWTB passieren. RTS zur aufrufenden Routine von der jeweiligen RWTB aus.

$D0F3-$D139 **$40/$41: ALLOCATE/DEALLOCATE INTERRUPT**
Aufruf von $D057 mit $01/$02 anstelle von $40/$41 im Akku.

$D0F8-$D123 $40 – ALLOCATE: Die Tabelle mit den Startadressen der Interrupt-Services in der Global Page wird nach einem freien Eintrag abgesucht. Falls kein Eintrag mehr frei ist, ERR #25: INTERRUPT TABLE FULL und Exit (=> $D078) via SET SYSERR ($BF09).
Ansonsten Eintrag in die Tabelle bei gleichzeitiger Überprüfung der Startadresse, die nicht kleiner als $100 sein darf (ERR #57: INVALID PARAMETER), SET SYSERR => $D078.

$D411 Die Nummer des gefundenen freien Eintrags wird in den Parameterblock des „user"-Programms gespeichert. RTS (=> $D078).

$D124-$D139 $41 – DEALLOCATE: Die Reference Number aus dem Parameterblock des „user"-Programms wird auf den zulässigen Bereich (1-4) geprüft (ERR #53: INVALID PARAMETER), SET SYSERR => $D078.
Die entsprechende Service-Startadresse wird auf 00 00 gesetzt. Ein Test, ob diese Startadresse überhaupt belegt war, findet nicht statt. RTS => $D078.

$D13A-$D1D9 **INTERRUPT SERVICE HANDLER**
Der IRQ-Vektor auf Seite 3 ($03FE/FF) zeigt auf $BFEB. Dort wird Bank 1 angeschaltet und hierher gesprungen.
Sämtliche Register werden in der Global Page zwischengespeichert.

$D156 Die Slotadresse ($C1-$C7) einer im Moment eventuell aktiven Karte wird auf $D1C4 gespeichert, um diese Karte nach dem Service wieder anzuschalten.

$D15F Wenn nicht mehr genügend Platz auf dem Stack vorhanden ist, werden $10 Bytes heruntergeholt und zwischengespeichert. Hier liegt eine Möglichkeit zum Systemabsturz (s. „Spezielles").

$D168 Die Speicherstellen $FA-$FF werden zwischengespeichert.

$D172 Die Service-Routinen, die in der Startadressen-Tabelle auf der Global Page definiert sind, werden der Reihe nach aufgerufen. Sie müssen mit gesetztem Carry zurückkehren, wenn die ihnen zugeordnete Interrupt-Quelle nicht den Interrupt ausgelöst hat. In diesem Fall wird die nächste (besetzte) Service-Routine aufgerufen. Kehrt eine Service-Routine mit gelöschtem Carry zurück, erfolgt ein Sprung zu $D19F.

$D19A Keine der Service-Routinen fühlt sich für den ausgelösten Interrupt zuständig, d.h. der Interrupt kam aus einer unbekannten Quelle. SYSTEM DEATH mit ERR #1.

$D19F Service erfolgreich beendet: $FA-$FF werden auf die Zeropage zurückgespeichert. Wenn auf dem Stack Platz gemacht werden mußte, werden diese Returnadressen ebenfalls zurückgeladen. X- und Y-Register werden wiederhergestellt.

$D1C2 Eine eventuell vorher angeschaltete Karte wird wieder aktiviert.

$D1CB Sprung zu INTEXIT auf der Global Page: Wiederherstellung der Konfiguration (Banks, ROM).

$D1CE JSRs zu den Service-Routinen

$D1DA **SET SYSERR**
Wird von $BF09 angesprungen (JMP). Setzt SYSERR und das Carry, holt eine Returnadresse vom Stack und kehrt zurück. Folge: Wenn eine Subroutine JSR $BF09 ausführt, ist die Subroutine damit auch beendet.

$D1E4 **SYSTEM DEATH** – 2. Einsprung. Setzt ERR# = 0 (s. „Global Page", $BF03/$BFF6).

$D1E6-$D23B **SYSTEM DEATH** – Einsprung mit ERR# im Akku.
Wird via $BF0C angesprungen. Setzt TEXT und gibt die Meldung: „INSERT SYSTEM DISK AND RESTART" aus. Text in ASCII auf $EFDF.

$D209 Ist die ERR# <> 0, folgt danach noch „-ERR" und die Nummer.

$D239 JMP $D239: Endlosschleife - Ende von SYSTEM DEATH.

$D23C-$D27E **HANDLER FÜR $Cx-$Dx - COMMANDS**
Wird von $D075 mit Index im Bereich $00-$13 aufgerufen.

$D23F Bestimmung der Routinen, die vor dem eigentlichen Kommando ausgeführt werden, durch eine weitere Tabelle auf $EF8D. Die unteren 5 Bits des durch die Tabelle erhaltenen Bytes bestimmen den Index in der (nächsten und letzten) Tabelle auf $EF65, die die Startadressen der Kommandos enthält. Die Startadressen werden ermittelt und in $F0B5/B6 gespeichert.
Passiert zwischen diesem Punkt und dem Sprung zum tatsächlichen Command ein Interrupt und muß dabei der Stack „entladen" werden, stürzt das System ab ($F0B5/B6 werden überschrieben, s. „Spezielles").

$D255 Das „Backup"-Bit des MLI ($F074) wird gesetzt (s. „Spezielles").

$D25C Bit 7 des Bytes aus der Tabelle in $EF8D war gesetzt: Aufruf von „Copy & Verify Pathname" ($D27F). Bei Fehler: Abbruch via SET SYSERR => $D078.

$D266 Bit 6 des Bytes aus der Tabelle in $EF8D war gesetzt: Aufruf von „Set File Data" ($D3C5). Bei Fehler: Abbruch via SET SYSERR => $D078.

$D270 Bit 5 des Bytes aus der Tabelle in $EF8D war gesetzt: Aufruf von SET TIME ($BF06).

$D273 Aufruf des eigentlichen Kommandos, dessen Startadresse in $F0B5/B6 gespeichert wurde, danach RTS zu $D078, falls kein Fehler aufgetreten ist.

$D278 Aufruf von SET SYSERR. Kehrt zu $D078 mit gesetztem Carry zurück.

$D27C Indirekter Sprung zur Befehlsausführung ($F0B5/B6).

$D27F-$D329 **COPY & VERIFY PATHNAME**
Der angegebene Pathname wird in den Bereich $F100.. kopiert und überprüft. Beginnt der Path mit „/", wird er als „vollständig" behandelt, d.h. ein eventuell gesetztes PREFIX wird nicht berücksichtigt.
Beim Kopieren wird auf Großbuchstaben übersetzt. Der auf $F100 gespeicherte Name hat danach die Form:
(Länge erster Name) „NAME1" (Länge zweiter Name) „NAME2" ... 00.
Wichtige Speicherstelle: $F07C (FULLNAME)

$D27F Der Pointer zum Pathname wird aus dem Parameterblock des „user"-Programms geholt und auf der Zeropage gespeichert.

$D28A Einsprung für RENAME: Der Pointer ist bereits (auf den neuen Pathname) gesetzt.
Möglicher Fehler: ERR #40: INVALID PATHNAME => RTS mit gesetztem Carry.

$D32A-$D381 **$C6 – SET PREFIX COMMAND**
Wichtige Speicherstellen:
$F019: Unitnummer; $F060/61: Blocknummer der neuen Prefix-Directory; $BF9A: PREFIX in der Global Page.

$D32A Copy & Verify Pathname ($D27F) wird erst hier aufgerufen. Grund: SET PREFIX ist auch mit dem Argument „/" (d.h. CLEAR PREFIX) möglich. Kehrt Copy & Verify Pathname mit ERR#40 zurück und ist die Länge des neuen Pathname = 0, wird das Prefix gelöscht ($BF9A = 0).

RTS => $D078. Ist die Länge des neuen Pathname im Fehlerfall <> 0, RTS mit gesetztem Carry (und ERR#40).

$D339 GET FILE ($D798) wird aufgerufen, um das entsprechende Volume (+ evtl. die entsprechende Subdirectory) zu finden und die zugehörigen Parameter zu laden. Die Fehlermeldung ERR#40 von GET FILE für den Fall, daß die Suche in einer Volume Directory endet (nur ein Name angegeben), wird maskiert, sie führt nicht zum Abbruch. (In diesem Fall liefert FIND FILE ENTRY den Storage Type „Subdirectory" zurück).

$D345 Ist der Type des gefundenen Files nicht „Subdirectory", ERR# 4B: UNSUPPORTED STORAGE TYPE.

$D35D PREFIX auf der Global Page wird neu gesetzt, die Unitnummer und die Blocknummer der durch das neue Prefix gesetzten (Sub)directory werden gespeichert.

$D372 Je nachdem, ob der neue Pathname vollständig war oder nicht, ersetzt er das alte Prefix bzw. wird daran angehängt. Es wird so kopiert, daß das Prefix immer auf $F1FF endet. Der Anfangsindex in der Seite $F100 ist in PREFIX ($BF94) gespeichert (s. „Programmiertricks"). RTS => $D078

$D37E ERR #4B: UNSUPPORTED STORAGE TYPE. RTS mit Carry gesetzt => $D078.

$D382-$D3C4 **$C7 – GET PREFIX COMMAND**

Die vom „user"-Programm angegebene Zieladresse wird aus dem Parameterblock geholt und auf der Zeropage gespeichert. Es wird ein maximaler Platzbedarf von $40 Zeichen angenommen, der damit belegte Speicherbereich wird gegen die SBM geprüft ($EE6C).

$D3A2 Da das Prefix immer auf $F1FF endet, ist der Startindex innerhalb der Seite $F100 variabel und muß zuerst berechnet werden.

$D3A9 Als erstes Byte wird die Länge des Prefix, danach das Prefix selber in den Zielbereich kopiert. Dabei findet eine Rückumwandlung der einzelnen Längenbytes in „/" statt.

$D3B9 Am Ende wird ein letzter „/" angehängt. RTS => $D078.

$D3C5-$D429 **SET FILE DATA**

Ausgehend von der File Reference Number aus dem Parameterblock des „user"-Programms werden die Startadresse des Filepuffers, die Unitnummer aus dem FCB-Eintrag und der Index innerhalb des FCB und VCB für einen File gesetzt.
Wichtige Speicherstellen:
$F051: VCB-Index; $F052: FCB-Index;
$004A/4B: Filepuffer – Datenbereich ($200 Byte);
$0048/49: Filepuffer – Indexblock-Bereich ($200 Byte).

$D3C5 Die File Reference Number wird aus dem Parameterblock des „user"-Programms geholt und auf den zulässigen Bereich (1-8) hin überprüft. Ist die Reference Number außerhalb des zulässigen Bereichs oder existiert für diese Reference Number kein FCB-Eintrag, ERR#43: INVALID REFERENCE NUMBER, RTS mit Carry gesetzt.

$D3E0 Einsprung mit bereits gesetztem FCB-Index für FLUSH(ALL), READ (nach direktem Blocklesen) und OPEN.
Die Pufferadresse wird aus der Global Page geholt ($EE26). Falls kein Puffer exisitert, Sprung zu $D412. Ansonsten werden die Startadressen des Datenteils und des Indexteils ($200 darüber) gesetzt.

$D3FE Der VCB wird nach dem Volume (Unit) für diesen File abgesucht. Wenn das Unit gefunden wurde, Sprung zu $D417.

$D407 Erhöhung des Index im VCB: nächster Eintrag, wenn der VCB noch nicht zu Ende ist.

$D40D Kein VCB-Eintrag für das Unit dieses Files, d.h. der File existiert für das MLI, aber es existiert kein Unit dafür: SYSTEM DEATH mit ERR #0A.

$D412 Es existiert kein Filepuffer für diesen File. SYSTEM DEATH mit ERR# 0B.
Diese beiden Fehler können eigentlich nicht vorkommen.

$D417 Wenn der gefundene VCB-Eintrag ungültig ist, Sprung zu $D407.

$D41F Alle Parameter gesetzt. RTS zur aufrufenden Routine.

$D421 Exit mit ERR#43: INVALID FILE REFERENCE NUMBER.
– Programmierfehler (s. „Spezielles").

$D42A-$D4F6 **$C5 – ONLINE COMMAND**
Die Zieladresse wird aus dem Parameterblock des „user"-Programms geholt und auf die Zeropage kopiert ($E403).
Wenn die angegebene Unitnummer = 0 ist (ALL), wird ein Platzbedarf von $100 Bytes, ansonsten $10 Bytes angenommen.

$D447 Der benötigte Speicherbereich wird gegen die SBM geprüft ($EE6C). Bei Fehler RTS mit Carry gesetzt => $D078.

$D44E Der Zielbereich wird gelöscht.

$D45F Für ONLINE mit angegebener Unitnummer: Sprung zu $D480.

$D461 ONLINE ALL: Alle Units werden aus der DEVICE TABLE in der Global Page nach $F08A.. kopiert ($DA57) und der Reihe nach durch ONLINE SPECIFIC bearbeitet.
Tritt bei einem Unit ein Fehler auf, wird nicht abgebrochen, sondern die Fehlernummer zwischengespeichert und nach Abschluß des Loops wieder geladen. RTS => $D078.

$D480 ONLINE SPECIFIC: Bearbeitung eines Units
Aufruf mit der Unitnummer im Akku. Im VCB wird ein Eintrag gesucht, der entweder diese Unitnummer enthält oder unbelegt ist ($DA69).

$D488 Die Volume Directory dieses Units wird (auf $F600-$F7FF) gelesen. War der Eintrag im VCB frei, d.h. ist dieses Unit noch nie gelesen worden, wird der VCB-Eintrag gemacht. Der etwas verzwickte Ablauf der folgenden Prüfungen ist in „Logische Abläufe" beschrieben.

$D4B8 ONLINE ERROR: Alle während ONLINE auftretenden Fehler führen zu einem Sprung hierher. Storage Type & Länge des Volume-Namens sowie Fehlernummer werden in den Zielpuffer gespeichert, gefolgt von

der Unitnummer, falls der Fehler #57: DUPLICATE VOLUME aufgetreten ist.
RTS entweder zu $D078 mit der Fehlernummer oder zum Loop von ONLINE ALL, wo die Fehlernummer zwischengespeichert wird.

$D4D6 Fehlerfreies ONLINE SPECIFIC: Der inzwischen im VCB eingetragene Name des Volumes wird in den Zielpuffer des „user"-Programms gespeichert.

$D4EB (Wird auch von ONLINE ERR mitbenutzt)
Zu der Namenslänge wird die Unitnummer addiert (höheres Nibble) und im Zielpuffer gespeichert. RTS entweder zu ONLINE ERR, zu $D078 oder zum Loop von ONLINE ALL.

$D4F7-$D691 **$C0 – CREATE COMMAND**
Vorher wurde Copy & Verify Pathname sowie SET TIME aufgerufen.

$D4F7 Der File wird gesucht (FIND FILE ENTRY: $D7AB). Wenn er gefunden wird, ERR#57: DUPLICATE FILENAME. RTS => $D078
Tritt dabei ein anderer Fehler als FILE NOT FOUND auf, ebenfalls RTS => $D078 mit entsprechend gesetzter ERR#.

$D504 Der vom „user"-Programm spezifizierte Storage Type wird aus dem Parameterblock geholt. Der Storage Type muß 0(!)/1/2/3 oder $0D (Subdirectory) sein, sonst ERR #4B: UNSUPPORTED STORAGE TYPE.

$D513 Das Unit, auf dem der File CREATEd werden soll, wird aktiviert und WRITE PROTECTED geprüft ($E65B).

$D518 Beim Suchvorgang nach einem File wird gleichzeitig nach einem freien Eintrag in der (hierachisch letzten) Directory gesucht. Existiert ein freier Eintrag, Sprung zu $D5B6, ansonsten weiter mit $D524.

$D520 Error Exit mit ERR #4B: UNSUPPORTED STORAGE TYPE. RTS => $D078.

$D524 **APPEND BLOCK TO SUBDIRECTORY**
Wenn die Directory, die zuletzt im Suchprozeß geladen wurde, keinen Parent Pointer hat, muß es sich um die Volume Directory handeln. ERR#49: VOLUME DIRECTORY FULL. RTS => $D078.

$D530 Die Nummer des zuletzt geladenen Blocks (= der letzte Block dieser Directory) wird gespeichert.
Der nächste freie Block auf dem Unit wird bestimmt (ALLOCATE NEXT BLOCK: $DCA9), die Blocknummer wieder zurückgespeichert. Die Nummer des nächsten freien Blocks wird in den (momentan noch geladenen und bis jetzt letzten) Block der Subdirectory als Forward Reference eingetragen. Der Block wird zurückgeschrieben ($DDDD).

$D54E Die Blocknummer dieses Blocks landet als Backward Reference im neuen Directoryblock, die gerade als Forward Reference eingetragene Nummer auf den RWTB-Parametern.

$D55D Der neue Block im Speicher wird bis auf die eben eingetragene Backward Reference gelöscht. Danach wird er auf das Unit geschrieben ($DDDD).

$D56D Der Block, in dem diese Subdirectory als Eintrag steht, wird gelesen ($DDE1), der Eintrag via Entrynummer herausgesucht.

$D590 Die Anzahl der von der Subdirectory belegten Blocks wird um $01, der EOF um $200 erhöht und der Block mit dem Eintrag zurückgeschrieben. Ein Update bis in den Key Block der Volume Directory ist hier nicht nötig, er erfolgt, wenn der Eintrag des Files, der CREATEd werden soll, gemacht wird!

$D5A6 Sprung zu $D4F7, kompletter Restart von CREATE.

$D5A9 Löschen des gesamten Directory Space ($F600-$F7FF). RTS (nur von $D5B6 aufgerufen).

$D5B6 Erstellung eines Subdirectory Key oder eines leeren Datenblocks.
Der gesamte Directory Space wird gelöscht ($D5A9).
Ein Eintrag wird vorbereitet: Aus dem „user"-Parameterblock wird CREATION DATE & TIME geholt. Falls nicht spezifiziert (0), wird die System Time eingesetzt.

$D5D9 Die Storage Types 0/1/2/3 werden alle zu Type 1 (!), Type $0D (Subdirectory) bleibt.

$D5E4 Zu dem Storage Type kommt (im selben Byte) die Länge des letzten Namens im Path; diese Bytes werden von $F100 nach $F01F kopiert.

$D5FC ACCESS, FILE TYPE, AUX TYPE werden aus dem Parameterblock des „user"-Programms, VERSION von $EFB0/B1 kopiert. Die Anzahl der vom neuen File belegten Blocks wird auf 00 01 gesetzt, und die Blocknummer des Key Blocks der Directory, in dem dieser File verzeichnet werden soll, kommt dazu. Damit ist ein kompletter Parametersatz für einen File-Eintrag (File Info) vorhanden.

$D630 Ist der neue File keine Subdirectory, Sprung nach $D66E.
Ansonsten wird ein Subdirectory Key Block im Directory Space erstellt. (Für ein Datenfile bleibt dieser Platz leer).

$D66E Der nächste freie Block auf dem Unit wird bestimmt und in der VBM belegt (ALLOCATE NEXT BLOCK: $DCA9).
Der neue Block wird geschrieben ($DDDD). Er enthält entweder einen Subdirectory Key Block oder nur Nullen.

$D682 Die Anzahl der aktiven Einträge in der Parent Directory wird um eins erhöht.
ENTRY & UPDATE ($D68A) sucht die Parent Directory heraus, macht den Eintrag und arbeitet sich „rückwärts" bis zum Key Block der Volume Directory hinauf.

$D68F Sprung zu $DD86: REWRITE VBM. RTS zu $D078 von da aus.

$D692-$D6AA Berechnung der Startadresse eines Eintrags im Directory Space ($F600-$F7FF). F01E enthält die Eintragsnummer, $F011 die Länge eines Eintrags. Setzt $0048/49. RTS zur aufrufenden Routine.

$D6AB-$D778 **ENTRY & UPDATE**

$D6AB Die System Time wird von der Global Page in den Parametersatz des neuen Eintrags (MODIFICATION DATE & TIME) kopiert.

$D6BB Einsprung mit bereits gesetzter DATE & TIME für SET F'INFO und RENAME. Bei diesen beiden Befehlen wird MODIFICATION DATE & TIME nicht neu gesetzt.
Der Directoryblock, in dem der Eintrag steht, wird gelesen. Die Startadresse des Eintrags wird berechnet ($D692) und der Eintrag wird vollständig mit den neuen Daten überschrieben. Ist der Block, der momentan bearbeitet wird, nicht der Key Block der Directory, wird er zurückgeschrieben ($DDDD) und der Key Block wird gelesen ($DDE1). Die (nur von CREATE/DESTROY veränderte) Anzahl der aktiven Einträge wird kopiert, genauso der nie veränderte ACCESS dieser Directory. Der Directory-Block wird zurückgeschrieben.

$D720 Wenn dieser Block der Key Block der Volume Directory war, RTS.
Ansonsten wird der Block gelesen, in dem diese Directory als Eintrag steht (Parent Block), die Position des Eintrags wird berechnet ($D692), MODIFICATION DATE & TIME werden auf den neuesten Stand gebracht (wenn eine System Time auf der Global Page existiert), der Block wird zurückgeschrieben. War dieser Block der Key Block der Directory, Sprung zu $D720. Ansonsten wird der Key Block gelesen (nötig zur Bestimmung der Parent Directory) und Sprung zu $D720.

$D778 RTS zur aufrufenden Routine (bei Fehler Carry gesetzt, sonst gelöscht).

$D779-$D77C Error exit: ERR #52: NOT A PRODOS VOLUME. SEC/RTS.

$D77D-$D797 Test, ob der gerade geladene Key Block ein ProDOS Key Block ist. Könnte auch für Subdirectories verwendet werden, wird aber nur für Volume Directories (vor Eintrag in den VCB) verwendet. RTS zum aufrufenden Programm (nur $DACF).

$D798-$D7AA **GET FILE**
Der Eintrag wird gesucht (FIND FILE ENTRY: $D7AB).

$D79D Einsprung für FLUSH – der entsprechende Directoryblock ist bereits geladen, der Pointer zum Eintrag ($0048/49) bereits gesetzt.
Die Daten des Eintrags (File Info) werden aus der Directory in den Variablenbereich kopiert. RTS zur aufrufenden Routine.

$D7AB-$D89F **FIND FILE ENTRY**
Das Volume wird gesucht und der Key Block der Volume Directory (oder der Subdirectory, bestimmt durch das System Prefix) geladen ($D92F).

$D7B2 Wenn der Path nur aus einem Namen besteht und somit hier endet, wurde jetzt die Volume Directory geladen und es wird kräftig „getürkt": Die Parameter, die sich auch in der Volume Directory befinden (DATE & TIME, ACCESS...) werden kopiert, danach wird mit Hilfe einer Tabelle auf $EFA8 ein Parametersatz für einen File / eine Subdirectory(!) erstellt (EOF, Anzahl der belegten Blocks etc.). Diese Angaben sind in der Volume Directory nicht enthalten, sie werden aber für OPEN und READ gebraucht. Danach wird der Storage Type ebenfalls auf

„Subdirectory" ($0D) gesetzt, bevor die Routine mit ERR#40: INVALID PATHNAME zur aufrufenden Routine zurückkehrt.

$D7DA Der Pathname ist noch nicht zu Ende: Wir befinden uns momentan im Key Block der (nächsten) (Sub)directory. Bei der Suche nach dem nächsten Namen im Path wird auch gleich noch nach einem freien Eintrag gesucht (s. CREATE). Momentan wurde noch kein freier Eintrag gefunden, das Flag dafür wird zurückgesetzt.

$D7E0 Der momentane Directoryblock wird nach dem Eintrag (nächster Name im Path) abgesucht, die Anzahl der gefundenen aktiven Einträge wird dabei mitgezählt, der erste freie Eintrag wird festgehalten ($D8D8). Wird der gesuchte Eintrag gefunden, Sprung zu $D84F. Bleiben noch aktive Einträge übrig (Angabe im Key Block), wird der nächste Block der Directory geladen. Sprung zu $D7E0.
Werden mehr Einträge gezählt, als vorhanden sein sollten, oder existiert keine Forward Reference zum nächsten Block, obwohl noch Einträge vorhanden sein sollten, ERR #51: DIRECTORY STRUCTURE DAMAGED und RTS zur aufrufenden Routine.

$D823 Wenn alle aktiven Einträge ordnungsgemäß heruntergezählt wurden, aber der gesuchte Eintrag nicht dabei war:
Wurde „unterwegs" kein freier Eintrag gefunden und ist dies nicht der letzte Block der Directory, wird gesetzt: freier Eintrag ist Eintrag Nummer eins im nächsten Block. Dieser „Patch" wird immer ausgeführt, ist aber nur für CREATE sinnvoll.
Wenn der jetzt nicht gefundene Name der letzte im Path war, wird ERR#46: FILE NOT FOUND, ansonsten ERR#44: DIRECTORY NOT FOUND gesetzt. RTS zur aufrufenden Routine.

$D84F Eintrag gefunden. Der Name wird im Path übersprungen ($D969).

$D852 Wenn der Path jetzt zu Ende ist, Sprung zu $D8C0.
Ansonsten muß dieser Eintrag für eine weitere Subdirectory stehen. Die Nummer des Key Blocks wird aus dem Eintrag geholt und der Key Block wird gelesen. Die Anzahl der aktiven Einträge wird aus dem gelesenen Key Block kopiert, der Key Block wird „überprüft", ob es sich wirklich um eine Subdirectory handelt.
Nach erfolgreicher Überprüfung werden Eintragslänge, Anzahl der Einträge und die Nummer des Parent Blocks kopiert ($D8A0). Sprung zu $D7DA: Absuchen dieser Directory nach dem nächsten Namen im Path.

$D8A0-$D8FB Kopiert die Key Block-Daten aus dem Directory Space. Wenn die geladene Directory eine Subdirectory ist, werden zusätzlich die Daten für den Parent Block kopiert. RTS zum aufrufenden Programm.

$D8C0 Eintrag gefunden: die Nummer des Eintrags und die Nummer des Blocks, in dem der Eintrag gefunden wurde, werden zwischengespeichert. RTS zur aufrufenden Routine. Wenn von $D852 her angesprungen, ist dies das Ende von FIND FILE ENTRY.

$D8D8-$D92E Sucht einen Eintrag (Name im Path) im momentan geladenen Directoryblock und hält den ersten freien Eintrag (Nummer + Flag) in diesem Block fest.
RTS mit Carry gelöscht, wenn Eintrag gefunden, ansonsten mit Carry gesetzt. Wird nur von $D7E5 aufgerufen.

$D92F-$D968 **GET VOLDIR/PREFIX DIRECTORY KEY BLOCK**
Das Volume wird gesucht und die Volume Directory oder die Prefix Directory wird geladen ($D988).
Aus dem geladenen Key Block werden Parameter kopiert ($D8A0), TOTAL BLOCKS (Blockanzahl) und die Nummer des ersten VBM-Blocks werden aus dem VCB kopiert, sie sind in einer Subdirectory (Prefix!) nicht enthalten.
RTS via $D969 zu $D7AB (nur von da aus aufgerufen).

$D969-$D96F Der momentane Pathname wird übersprungen und der Index des nächsten Namens gesetzt. Ist der Pathname hier zu Ende, RTS mit Z-Flag gesetzt („EQ").

$D970-$D987 Der Index des nächsten Pathname wird berechnet, aber nicht neu gesetzt. Ist der Pathname hier zu Ende, RTS mit Z-Flag gesetzt.

$D988- **SEARCH VOLUME/PREFIXED DIRECTORY**
Wenn ein vollständiger Pathname angegeben wurde, Start am Anfang des Namens, ansonsten Start am Anfang des System Prefix.

$D999 Der VCB wird nach dem ersten Namen im Path (s.o.) abgesucht. Wird der Name nicht gefunden, Sprung zu $D9FD.

$D9C0 VCB-Eintrag gefunden: der VCB-Index, die Unitnummer und die Blocknummer 00 02 (Volume Directory) werden gesetzt.

$D9D4 War der Pathname vollständig, bleibt es dabei, ansonsten wird die Blocknummer der Prefix Directory gesetzt.

$D9EA Der Key Block wird gelesen ($DDE1). Es wird überprüft, ob es sich dabei um einen ProDOS Key Block handelt und ob die Namen übereinstimmen ($DA91). Wenn ja, RTS zur aufrufenden Routine (nur $D92F).

$D9F5 Der Name der gefundenen Directory ist nicht der erwartete (oder kein ProDOS-Volume): Wenn das eigentlich erwartete Volume einen offenen File enthält, RTS mit VOLUME NOT FOUND zu $D92F.

$D9FD Einsprung, wenn das gesuchte Volume nicht im VCB gefunden wurde.
Alle in der Global Page verzeichneten Units (Devices) werden kopiert ($DA57). Wurde das Volume nicht im VCB gefunden, erfolgt die Suche von oben nach unten in der Device-Tabelle, ansonsten wird das Unit, auf dem gerade bereits gesucht wurde, zuerst behandelt.

$DA0B Start des Loops zum Absuchen („ONLINE") aller Units, bis das gesuchte Volume gefunden wurde.

$DA16 Das letzte Unit wurde (erfolglos) abgesucht:
RTS mit ERR#45: VOLUME NOT FOUND zu $D92F.

$DA25 Die Nummer des momentanen Units wird gesetzt und das Unit aus der kopierten Tabelle gestrichen. Der Eintrag für dieses Unit oder ein freier Eintrag wird im VCB gesucht ($DA69). Wird kein Eintrag für dieses Unit und auch kein freier Eintrag mehr gefunden, RTS mit Carry gesetzt. Wird ein Eintrag gefunden und hat dieses Unit einen offenen File, nächstes Unit (Sprung zu $DA0B).

$DA3F Der Key Block der Volume Directory dieses Units wird gelesen. Die Daten werden im VCB eingetragen ($DAB7). Durch Aufruf von $DA91 wird verglichen, ob diese Volume Directory den gesuchten Namen hat. Wenn ja, RTS, ansonsten Sprung zu $DA0B.

$DA57-$DA68 Kopiert alle Devices (Units) von der Global Page. RTS zur aufrufenden Routine.

$DA69-$DA90 Sucht einen freien Eintrag oder den Eintrag für ein bestimmtes Unit im VCB via Unitnummer.

$DA76 Wird ein VCB-Eintrag für dieses Unit gefunden, wird der VCB-Index gesetzt, RTS mit Carry gelöscht zur aufrufenden Routine.

$DA7B Wenn der momentane Eintrag nicht frei ist, nächster Eintrag.
War das der letzte Eintrag, und wurde „unterwegs" auch kein freier Eintrag gefunden, RTS mit Carry gesetzt und ERR#55: VCB TABLE FULL, ansonsten ist der VCB-Index auf den freien Eintrag gesetzt. RTS mit Carry gelöscht.

$DA91-$DAB6 Überprüfung, ob der geladene Block ein ProDOS Key Block ist und Vergleich mit dem momentanen Namen im Path. RTS mit Carry gelöscht bei erfolgreicher Überprüfung, ansonsten mit Carry gesetzt zur aufrufenden Routine.

$DAB7-$DB1B Eintrag eines (neuen) Volumes in den VCB
Wenn der VCB-Eintrag bereits besetzt war, wird zuerst verglichen, ob der Volume-Name derselbe geblieben ist ($DB1C). Wenn ja, RTS.

$DAC4 Einsprung für ONLINE: es wurde bereits geprüft, ob der VCB-Eintrag besetzt war (und ob offene Files auf diesem Unit existieren).
Der (neue) VCB-Eintrag wird gelöscht. Damit werden auch sämtliche Flags (OPEN file etc) zurückgesetzt.

$DACF Es wird festgestellt, ob der geladene Block ein ProDOS Volume Directory Key Block ist ($D77D), danach wird der VCB nach einem Eintrag mit demselben NAMEN abgesucht ($DB3D). Wird ein Eintrag (kann ein anderes Unit sein!) mit demselben Namen (ohne offene Files darauf) gefunden, wird dieser Eintrag ungültig gemacht.
Enthält ein anderer Eintrag mit demselben Namen dagegen einen offenen File, wird der neue Eintrag nicht gemacht!

$DADC Die Parameter werden aus dem Block in den VCB-Eintrag kopiert. RTS zur aufrufenden Routine.

$DB1C-$DB3C Vergleich der neu geladenen Volume Directory mit einem VCB-Eintrag via Name (nicht Unitnummer!). RTS mit Carry gelöscht, wenn gefunden, ansonsten mit Carry gesetzt zur aufrufenden Routine.

$DB3D-$DB65 Absuchen des gesamten VCB nach dem Namen der gerade geladenen Volume Directory. Benutzt $DB1C. Wird der Name gefunden und hat das Unit keinen offenen File, wird der gefundene Eintrag ungültig gemacht.
RTS mit Carry gelöscht, falls das Volume nicht im VCB gefunden wurde oder das gefundene Volume keinen offenen File hat.
Falls dagegen das Volume mit einem offenen File gefunden wurde, wird der VCB-Index gespeichert und ein Flag gesetzt (nur für ONLINE von Bedeutung). RTS mit Carry gesetzt.

$DB66-$DBD7 **SPACE ON VOLUME?**
Die aufrufende Routine setzt vorher die Anzahl der benötigten Blocks sowie den VCB-Index für das Volume. Die VBM-Blocks werden nicht in den Bereich $F400-$F5FF, sondern in den Directory Space ($F600-$F7FF) geladen!

$DB66 Falls diese Kalkulation für dieses Volume schon einmal durchgeführt wurde, sind die Bytes $14/$15 im VCB-Eintrag entsprechend gesetzt (<> 0). Sprung zu $DBC3. Falls allerdings keine Blocks mehr auf diesem Volume frei sind, wird die Berechnung erneut durchgeführt.

$DB71 Die Anzahl der möglichen VBM-Blocks wird aus der Gesamtkapazität des Volumes bestimmt ($DC15) und das Ergebnis als Zähler gespeichert. Falls sich noch ein VBM-Block eines anderen Units im Speicher befindet, wird er (falls geändert) zurückgeschrieben ($DD86). Danach wird die Blocknummer des ersten VBM-Blocks (00 06) aus dem VCB-Eintrag geholt und gesetzt.

$DB96 Loop durch alle möglichen VBM-Blocks. Der Block wird gelesen ($DDE1), die Anzahl der freien Blocks innerhalb dieses VBM-Blocks wird berechnet ($DBD8). Falls dies nicht der letzte VBM-Block war, Erhöhung der Blocknummer um 1 (nächster „Extent") und Loop zu $DB96.
Wurde während des Loops kein einziger freier Block gefunden, ist das „no Space"-Flag ($F05B) immer noch gesetzt: RTS mit ERR#48: VOLUME FULL.

$DBAC Die Anzahl der gezählten freien Blocks wird im VCB-Eintrag gespeichert, das „no Space"-Flag ist auf die Extent-Nummer des ersten VBM-Blocks gesetzt, in dem ein freier Block gefunden wurde.

$DBC3 Die Anzahl der benötigten Blocks (gesetzt von der aufrufenden Routine) wird abgezogen. Ergibt sich dabei ein Wert < 0, RTS mit ERR#48: VOLUME FULL, ansonsten RTS mit Carry gelöscht, beide zur aufrufenden Routine.

$DBD8-$DC04 Zählt die Anzahl der freien Blocks in einem VBM-Block, der auf $F600-$F7FF geladen wurde. Benutzt $DC05. Das „no Space"-Flag ($F05B) wird auf die „Extent"-Nummer des ersten VBM-Blocks gesetzt, in dem ein freier Block gefunden wurde. RTS zur aufrufenden Routine (nur $DB9B).

$DC05-$DC14 Zählt die gesetzten Bits (= freie Blocks) in einem Byte. RTS zur aufrufenden Routine (nur von $DB71/$DBFA benutzt).

$DC15-$DC26 Berechnet die Maximalzahl der möglichen VBM-Blocks auf einem Volume minus 1 aus TOTAL BLOCKS aus dem VCB-Eintrag. RTS zur aufrufenden Routine.

$DC27-$DCA8 **DEALLOCATE BLOCK**
Aufruf mit Blocknummer in A-X. Der VBM-Block befindet sich auf $F400-$F5FF.

$DC2E Wenn die Blocknummer die höchste mögliche Blocknummer auf diesem Volume überschreitet, ERR#5A: FILE STRUCTURE DAMAGED.

$DC37 Aus der Blocknummer wird ein Index und das entsprechende Bit innerhalb eines VBM-Bytes berechnet, das wieder gesetzt wird (= Block frei) sowie die Nummer des VBM-Blocks, in dem der Block verzeichnet ist.

$DC57 Falls sich noch der VBM-Block eines anderen(!) Units im Speicher befindet, wird er zurückgeschrieben ($DD57).

$DC5C Befindet sich der benötigte VBM-Block im Speicher, Sprung zu $DC7A. Falls sich noch ein VBM-Block dieses(!) Units im Speicher befindet, hat der Aufruf von $DD57 diesen Block nicht zurückgeschrieben. Wird nun ein anderer VBM-Block dieses Units benötigt, wird zuerst sichergestellt, daß auch VBM-Blocks dieses Units nötigenfalls zurückgeschrieben werden ($DD86).

$DC75 Wenn sich der benötigte VBM-Block noch nicht im Speicher befindet, wird er geladen ($DD97).

$DC7A Der benötigte VBM-Block ist im Speicher. Aus der Blocknummer wird eine der beiden möglichen Speicherseiten berechnet und im vorher ermittelten Byte das entsprechende Bit wieder gesetzt (= Block frei). Die Anzahl der wieder freigegebenen Blocks wird um eins erhöht (Zähler für SET EOF und DESTROY). RTS zur aufrufenden Routine.

$DCA5 Error exit: ERR #5A: FILE STRUCTURE DAMAGED. RTS zur aufrufenden Routine mit Carry gesetzt.

$DCA9-$DD34 **ALLOCATE NEXT BLOCK**
Ein eventuell noch im Speicher befindlicher VBM-Block eines anderen Units wird zurückgeschrieben, der (erste) VBM-Block dieses Units wird gelesen ($DD57).

$DCAE Der „Page indicator" wird zurückgesetzt, die erste Speicherseite des geladenen VBM-Blocks wird abgesucht. Wird ein freier Block gefunden, Sprung zu $DCD2. Ansonsten wird der „Page indicator" erhöht und die zweite Hälfte des Blocks abgesucht. Falls dabei ein freier Block gefunden wird, ebenfalls Sprung zu $DCD2.

$DCC9 Kein freier Block in diesem VBM-Block: Die Maximalzahl der möglichen VBM-Blocks auf diesem Volume wird bestimmt, der nächste VBM-Block wird ggf. gelesen ($DD35). Sprung zu $DCAE, wenn ein weiterer VBM-Block gelesen wurde, ansonsten RTS mit Carry gesetzt und ERR#48: VOLUME FULL.

$DCD2 Freier Block gefunden: Aus der VBM-Nummer und der Fundstelle innerhalb des Blocks wird die Blocknummer berechnet und gespeichert, der Block wird belegt. Die Anzahl der freien Blocks auf diesem Volume wird im VCB-Eintrag um eins heruntergesetzt, das Flag „VBM altered" gesetzt. RTS zur aufrufenden Routine mit der Nummer des freien Blocks in A-Y. Der so geänderte VBM-Block wird erst zurückgeschrieben, wenn ein anderer VBM-Block geladen wird (oder bei FLUSH/CLOSE eines Files auf diesem Volume).

$DD35-$DD56 Stellt anhand des VCB-Eintrags (TOTAL BLOCKS) fest, ob auf diesem Volume noch weitere VBM-Blocks möglich sind. Der momentan im Speicher befindliche VBM-Block wird zurückgeschrieben, falls geändert ($DD86). RTS zur aufrufenden Routine (nur $DCCC) via $DD57 (Lesen des nächsten VBM-Blocks).

$DD53 Error Exit. ERR#48: VOLUME FULL (keine weiteren VBM-Blocks auf diesem Unit. RTS zu $DCCC mit Carry gesetzt.

$DD57- **GET VBM**
Wenn sich ein (beliebiger!) VBM-Block des momentanen Units bereits im Speicher befindet, Sprung zu $DD70.
Ein eventuell noch im Speicher befindlicher VBM-Block eines anderen Units wird zurückgeschrieben, wenn er verändert wurde ($DD86).

$DD70 Wurde hierher von $DD60 gesprungen, kann immer noch ein geänderter VBM-Block dieses Units im Speicher sein, der aber nicht der richtige Block sein muß (anderer „Extent"). Wurde auf diesem Block etwas verändert, wird der Aufruf von $DD97 (Read VBM) übersprungen, um diesen geänderten Block nicht zu überschreiben. Das Zurückschreiben dieses Blocks und das Laden des richtigen VBM-Blocks wird dann in den Routinen nachgeholt, die diese Routine aufrufen.

$DD7D Die Nummer (Extent) des momentanen VBM-Blocks wird mal 2 genommen und gespeichert. Sie dient zur Adressierung des zu belegenden/freizustellenden Blocks innerhalb des VBM-Blocks (= Blocknummer / $800). RTS zur aufrufenden Routine.

$DD86-$DD96 Falls das „VBM altered"-Flag gesetzt ist, wird der momentan im Speicher befindliche VBM-Block auf das dazugehörige Volume zurückgeschrieben ($DDD9) und das Flag zurückgesetzt. RTS zur aufrufenden Routine.

$DD97-$DDD8 **READ VBM**
Aufruf mit Unitnummer im Akku und gesetztem VCB-Index.

Aus dem VCB-Eintrag wird der momentane Extent geholt (0 für den ersten VBM-Block), zu der „Blocknummer VBM" (ebenfalls aus dem VCB-Eintrag) addiert und gespeichert. READ wird gesetzt.

$DDB4 Einsprung für WRITE VBM, WRITE ist bereits gesetzt.
Die Ziel/Quelladresse und die Blocknummer werden gesetzt, der Block wird gelesen/geschrieben ($DDE8). Die „Nummer des zuletzt benutzten Units" wird dabei zwischengespeichert und nachher wieder hergestellt. RTS zur aufrufenden Routine.

$DDD9 **WRITE VBM BLOCK**, setzt WRITE und springt zu $DDB4.

$DDDD **WRITE BLOCK** von $F600. Setzt WRITE und springt zu $DDE3.

$DDE1 **READ BLOCK** nach $F600. Setzt READ und lädt $F6 als höherwertigen Teil der Zieladresse.

$DDE8 Einsprung für READ nach $F400 (VBM)

$DDE9-$DE07 Gemeinsamer Code für WRITE/READ von/nach $F600 und READ nach $F400.
Nach Abschalten der Interrupts wird die Ziel/Quelladresse gesetzt, SYSERR auf Null gesetzt, die Unitnummer kopiert und $D0DA aufgerufen. RTS zur aufrufenden Routine mit Carry gelöscht/gesetzt und wieder hergestelltem INT-Status.

$DE08-$DE19 **$CF – GET MARK COMMAND**
Vorher wurde „Set File Data" ($D3C5) ausgeführt.
Mit dem von $D3C5 gesetzten FCB-Index wird die momentane Position aus dem FCB-Eintrag geholt und in den Parameterblock des „user"-Programms kopiert. RTS => $D078.

$DE1A-$DE1D Error Exit für SET MARK. ERR#4D: POSITION OUT OF RANGE. RTS mit Carry gesetzt => $D078.

$DE1E- **$CE – SET MARK COMMAND**
Vorher wurde „Set File Data" ($D3C5) ausgeführt.
Die „Position gewünscht" aus dem Parameterblock des „user"-Programms wird kopiert und mit dem momentanen EOF des Files (FCB-Eintrag) verglichen. Wird der EOF überschritten, Sprung zu $DE1A.

$DE3B Einsprung für alle internen Aufrufe.
„Position jetzt" und „Position gewünscht" werden verglichen. Ist die Differenz positiv und kleiner $200, Sprung zu BLOCK IN MEMORY ($DF7C).

$DE61 Nicht derselbe Block: ist der Storage Type des Files = 0, Sprung zu $DE6D. Für die Typen 1/2/3: Sprung zu $DE76, für alle anderen Storage Types: Sprung zu $DFAE.

$DE6D Der Versuch, SET MARK in einem File mit Storage Type 0 durchzuführen, endet damit, daß die Speicherstelle $F352 auf Null gesetzt wird. Exit mit Carry gesetzt und ERR#43: INVALID REFERENCE NUMBER => $D078 (s. „Spezielles").

$DE76 SET MARK für Files (Type 1/2/3).
Befindet sich noch ein geänderter Datenblock dieses Files im Filepuffer (STATUS-Bit 6), wird er zurückgeschrieben ($E087).
Ist die Differenz von „Position gewünscht" und „Position jetzt" größer als 01 FF FF, so wird ein neuer Indexblock benötigt, Sprung zu $DEBA.

$DEA6 Wenn die „Position gewünscht" kleiner als 00 02 00 ist, kann es sich immer noch um einen File mit Type 1 handeln (die „Position jetzt" wurde dann von OPEN oder SET EOF auf einen „unmöglichen" Wert gesetzt (s. dort)). Ist die „Position gewünscht" < 00 02 00, wird die „Nummer des ersten Blocks im File" gesetzt. Sprung zu $DF72. Ansonsten Sprung zu $DF0E: Erweiterung auf Type 2.

$DEBA Positionsdifferenz > 256 Blocks: Befindet sich noch ein geänderter Indexblock im Filepuffer (STATUS-Bit 7), wird er zurückgeschrieben ($E09B). Wenn der File bereits den Type 3 hat, wird keine Erweiterung benötigt, Sprung zu $DEF3.
Ist die gewünschte Position < 02 00 00, kann es sich noch um einen File mit dem Type 2 (oder sogar 1) handeln. Die Expansion-Bits in STATUS werden gelöscht. Hat der File den Type 1, Sprung zu $DEA6 (der Test auf Erweiterung zum Type 2 wird dort vorgenommen). Hat der File den Type 2, wird der Indexblock in den Filepuffer gelesen ($E02E), der FCB-Eintrag erhält die Blocknummer als „Indexblock jetzt im Speicher". Sprung zu $DF20.

$DEF3 Für Files mit Type 3: Die Expansion-Bits in STATUS werden zurückgesetzt und der Masterblock wird in den Indexbereich des Filepuffers gelesen. Die Blocknummer des benötigten Indexblocks wird ermittelt.

$DF0E Existiert kein Indexblock im Master (00 00) oder wird eine Erweiterung auf Type 2 benötigt, Sprung zu $DF39 mit „neuer Indexblock benötigt".

$DF13 Der Indexblock wird gelesen ($E010).

$DF20 Indexblock im Speicher (Typen 2 und 3).
Der benötigte Datenblock wird ermittelt. Hat der Indexblock keinen Eintrag für diesen Datenblock („sparse" oder EOF überschritten) wird Erweiterungsbit 0 gesetzt („neuer Datenblock benötigt"). Ansonsten Setzen der Blocknummer und Sprung zu $DF6C.

$DF39 – Neuer Master / Index / Datenblock benötigt
Einsprung mit gesetzten Erweiterungsbits im Akku.
Wird nur ein neuer Datenblock benötigt, wird allein der Datenbereich des Filepuffers gelöscht. Ansonsten wird der Indexbereich ebenfalls gelöscht ($DF5A). Sprung zu BLOCK IN MEMORY.

$DF5A-$DF9B Löschen des Datenteils im Filepuffer und RTS zur aufrufenden Routine.

$DF6C Der zweite Teil der Blocknummer wird aus dem Indexblock gelesen.

$DF72 Der zweite Teil der Blocknummer wird gesetzt, der Block wird in den Datenbereich des Filepuffers gelesen ($DFF7). Die Expansion-Bits von STATUS werden zurückgesetzt. Weiter mit $DF7C.

$DF7C BLOCK IN MEMORY
Die alte „Position jetzt" wird aus dem FCB-Eintrag kopiert, die neue (und jetzt erreichte) Position wird hineingeschrieben. Ein Pointer wird innerhalb des Datenpuffers auf die gewünschte Speicherseite gesetzt. RTS zur aufrufenden Routine oder zu $D078, wenn SET MARK als MLI-Command durchgeführt wurde.

$DFA2-$DFAD Setzt die Expansion-Bits in STATUS zurück. RTS zur aufrufenden Routine.

$DFAE SET MARK in Directories
Angesprungen von $DE6A: Storage Type ist > 03.
Wenn der Storage Type nicht $0D ist (auch für Volume Directories, s. FIND FILE ENTRY), RTS via SET SYSERR mit ERR#4A: INCOMPATIBLE FILE FORMAT zur aufrufenden Routine oder $D078 („user"-Command).

$DFB7 Je nachdem, ob die „Position gewünscht" größer oder kleiner als die „Position jetzt" ist, verzweigt die Ausführung zu $DFC6 oder $DFD4.

$DFC6 Suche rückwärts: Über die Backward Reference wird der nächste Block bestimmt und gelesen ($DFE2) bis die gewünschte Position erreicht ist. RTS via $DF7C zur aufrufenden Routine.

$DFD4 Suche vorwärts. Die Bestimmung der nächsten Block erfolgt über die Forward Reference. RTS via $DF7C.

$DFE2-$DFF6 Aufruf mit 0 oder 2 in Y. (= Backward/Forward Reference).
Der nächste Block wird bestimmt. Ist die Reference = 00 00, ist der Anfang/das Ende der Directory erreicht. RTS mit ERR#4C: EOF ENCOUNTERED zur aufrufenden Routine. Ansonsten weiter mit $DFF7 und RTS von dort aus.

$DFF7-$E00F Liest einen Block in den Datenteil des Filepuffers. Aufruf mit gesetzter Blocknummer. Der Block wird gelesen ($E054) und die Blocknummer als „Datenblock jetzt im Speicher" im FCB-Eintrag gesetzt. RTS zur aufrufenden Routine.

$E010-$E029 Liest einen Block in den Indexbereich des Filepuffers. Aufruf mit gesetzter Blocknummer. Der Block wird gelesen ($E054) und die Blocknummer als „Indexblock jetzt im Speicher" im FCB-Eintrag gesetzt.

$E02A-$E02D Schreibt den „ersten Block im File", d.h. einen Block aus dem Indexbereich des Filepuffers auf die im FCB-Eintrag angegebene Blocknummer (für Typen 2 und 3).

$E02E Einsprung für READ „erster Block im File" in den Indexbereich des Filepuffers (Typen 2 und 3).

$E030 Gemeinsamer Code für READ/WRITE „erster Block des Files" in/aus dem Indexbereich des Filepuffers. Die Blocknummer ergibt sich aus dem Wert des Y-Registers, das hier auf die Bytes $0C/$0D des FCB-Eintrags zeigt.

$E03A Einsprung für WRITE DATA BLOCK/WRITE INDEX BLOCK. Das Y-Register zeigt auf die FCB-Bytes $10/$11 (DATA) oder $0E/$0F (INDEX).
Ergibt sich dabei als Blocknummer 00 00, d.h. ist der FCB-Eintrag nicht besetzt, folgt SYSTEM DEATH mit ERR#0C.
Weiter mit $E054.

$E054-$E080 Allgemeine READ/WRITE-Routine zum Daten/Indexteil des Filepuffers. Aufruf/Einsprung mit gesetzter Blocknummer. Je nach Wert des X-Registers ($48 oder $4A) wird mit dem Index- oder Datenteil des Filepuffers gearbeitet. Die Unitnummer wird aus dem FCB-Eintrag geholt und gesetzt, der Block wird gelesen/geschrieben ($D0DA). Der INT-Status wird wieder hergestellt. RTS zur aufrufenden Routine mit Carry gesetzt bei Fehler, ansonsten Carry gelöscht.

$E081-$E086 Wenn der momentane VBM-Block verändert wurde, wird er zurückgeschrieben ($DD86). Sprung zu $E02A (WRITE des „ersten Blocks im File" aus dem Indexbereich des Filepuffers).

$E087-$E099 (RE)WRITE CURRENT DATA BLOCK (aus dem Filepuffer). Das Y-Register zeigt auf die FCB-Bytes $10/$11, das X-Register auf den Pointer $4A/$4B. Der Block wird geschrieben ($E03A). Bit 6 von STATUS wird via $E0AF gelöscht (Datenblock zurückgeschrieben). RTS zur aufrufenden Routine von da aus.

$E09B-$E0AD (RE)WRITE CURRENT INDEX BLOCK (aus dem Filepuffer). Das Y-Register zeigt auf die FCB-Bytes $0E/$0F, das X-Register auf den Pointer $48/$49. Der Block wird geschrieben ($E03A). Bit 7 von STATUS wird via $E0AF gelöscht (Indexblock zurückgeschrieben).

$E0AF-$E0BB Löscht Bit 6 oder Bit 7 in STATUS. RTS zur Routine, die (RE)WRITE CURRENT DATA/INDEX BLOCK aufgerufen hat.

$E0B9-$E1A8 **$C0 – OPEN COMMAND**
Vorher wurde Copy & Verify Pathname ($D27F) ausgeführt.

$E0B9 Der File wird gesucht ($D798). Kommt GET FILE mit ERR#40 zurück, wird dieser Fehler maskiert: es ist auch möglich, Volume Directories via OPEN zu lesen!
Alle anderen Fehler (FILE NOT FOUND etc.): RTS => $D078.
Der gesamte FCB wird nach diesem File abgesucht ($E1A9). Dabei werden (vorläufig) nur Files berücksichtigt, bei denen auch WRITE erlaubt ist. Existiert bereits ein Eintrag, ERR#50: FILE (ALREADY) OPEN => $D078.

$E0CB Die Routine auf $E1A9 hat gleichzeitig den Index für den ersten freien FCB-Eintrag festgehalten und ein Flag „freier Eintrag gefunden" gesetzt. Wurde kein freier Eintrag gefunden, ERR#42: MAXIMUM NUMBER OF FILES OPEN => $D078.

$E0D7 Der freie FCB-Eintrag wird vollständig gelöscht.

$E0E2 Von GET FILE gesetzte Daten (Unitnummer etc.) werden in den Eintrag kopiert.

$E0F3 Der Storage Type wird heruntergeshiftet und im FCB eingetragen. Die beiden LSBs von ACCESS (READ/WRITE) werden in den FCB kopiert. Für Files, die nicht den Type 1/2/3 haben (d.h. Directories), wird immer nur READ zugelassen!

$E110 Falls der File bereits OPEN war ($E1A9), aber READ ONLY, erfolgt der Abbruch mit FILE (ALREADY) OPEN erst hier nach dem erneuten Setzen von ACCESS (s. „Spezielles").

$E115 Die „min" und „max"-Versionsnummern des MLI und des Files werden verglichen, ERR#4E: INCOMPATIBLE FILE FORMAT => $D078, falls der File unter einer nicht passenden MLI-Version erzeugt wurde. Derselbe Fehler wird erzeugt, wenn der File weder den Type 1/2/3 noch den Type $0D hat (FIND FILE ENTRY setzt auch Volume Directories auf Type $0D, s. dort).

$E129 „Erster Block im File", EOF und Anzahl der belegten Blocks werden mit Hilfe einer Adress-Übersetzungstabelle in den FCB-Eintrag kopiert.

$E144 Die Nummer des FCB-Eintrags wird auf Byte $00 des Eintrags geschrieben. Erst damit ist der Eintrag gültig und „besetzt". Der Start des Filepuffers wird aus dem Parameterblock des „user"-Programms geholt und gegen die SBM geprüft ($EDD7). Bei Fehler Sprung zu $E173.

$E152 „LEVEL" ($BF94) wird von der Global Page in Byte $1B des FCB-Eintrags kopiert (s. „Spezielles").

$E158 Hat der File den Type $0D, Sprung zu $E18A. Ansonsten wird die „Position jetzt" auf FF 00 00 (= „unmöglich") und die „Position gewünscht" auf 00 00 00 gesetzt und es folgt ein Aufruf von SET MARK. Da STATUS durch das Löschen des gesamten FCB-Eintrags ($E0D7) den Wert 0 hat (nichts verändert), wirft GET MARK den momentanen Inhalt des Filepuffers (undefiniert!) hinaus und lädt den ersten Datenblock (+ Indexblock, je nach Storage Type) des Files in den Filepuffer. Passiert dabei kein Fehler, Sprung zu $E18F.

$E173 Error Exit: Wurde bereits ein Puffer für diesen File in der SBM belegt, wird er wieder gelöscht ($EE34). Der FCB-Eintrag wird ungültig gemacht. RTS mit Carry gesetzt und ERR# im Akku zu $D078.

$E18A Lesen des ersten Blocks für Directories: Der erste Block (Blocknummer aus dem Eintrag für die Directory selber) wird in den Datenbereich des Filepuffers gelesen ($DFF7). Der Indexbereich bleibt dabei undefiniert. Bei Fehler Sprung zu $E173, sonst weiter mit $E18F.

$E18F Der erste Datenblock des Files ist im Filepuffer. Die Anzahl der offenen Files für dieses Volume wird im VCB-Eintrag erhöht, das Flag „OPEN File(s)" (Byte $1C) wird gesetzt. Die FCB-Reference Number (nicht der Index!) wird im Parameterblock des „user"-Programms gespeichert. RTS => $D078.

$E1A9-$E1F6 Vergleicht, ob ein File bereits im FCB steht. Der Vergleich umfaßt sämtliche Daten außer Position, EOF und STATUS. (Ein Vergleich der Namen allein ist wegen der hierarchischen Struktur der Directories nicht ausreichend).
Gleichzeitig wird ein freier FCB-Eintrag gesucht, der erste freie Eintrag wird festgehalten (Index + Flag). Files, bei denen ACCESS im FCB nur auf READ gesetzt ist, werden ignoriert, allerdings wird ein weiteres Flag gesetzt, wenn der gesuchte File als READ ONLY bereits im FCB eingetragen ist.
RTS zur aufrufenden Routine mit CLC, wenn nicht gefunden oder READ ONLY, ansonsten mit Carry gesetzt.

$E1F7-$E2EE **$CA – READ COMMAND**
Vorher wurde „Set File Data" ($D3C5) ausgeführt.
Die Logik des 3-Stufen-Prinzips von READ ist in „Logische Abläufe" bereits ausführlich behandelt worden.

$E1F7 Die Zieladresse (Start) wird aus dem Parameterblock des „user"-Programms geholt und auf die Zeropage kopiert ($E403), die Länge des Puffers wird gesetzt ($E3E8), die „Position jetzt" + Request Count wird gegen den EOF des Files verglichen ($E415).

$E201 Wenn READ bei diesem File nicht erlaubt ist, ERR#4E: FILE ACCESS ERROR und Sprung zu $E2E8.

$E20A Wenn der READ über den EOF hinausgehen würde, wird die Anzahl der zu transferierenden Bytes entsprechend herabgesetzt. Bleibt dabei nichts übrig, ERR#4C: EOF ENCOUNTERED, Sprung zu $E2E8.

$E231 Ist der Request Count = 0 (nicht zu verwechseln mit dem durch EOF begrenzten Zähler!), RTS via $E2EF.

$E23C Der Zielbereich wird gegen die SBM geprüft ($EE6C), bei Fehler Sprung zu $E2E8.

$E248 Für Directory-Files: Sprung zu $E3B1.

$E24B Stufe 1: Byte-per-Byte
Mit SET MARK wird der Block in den Filepuffer geladen. Die Ziel- und Quellpointer werden justiert ($E306), die Bytes, die aus diesem Block gewünscht werden, werden kopiert ($E330).

$E256 Ist die Anzahl der zu transferierenden Bytes bis auf Null heruntergezählt oder wurde der Transfer durch NEWLINE beendet, Sprung zu $E2EF.

$E258 Ist NEWLINE aktiv, Loop zu $E24B. Bleibt weniger als ein ganzer Block zu transferieren übrig oder wurde dieser Block vorher mit WRITE verändert und muß noch zurückgeschrieben werden, ebenfalls Loop zu $E24B.

$E26D Stufe 2: Block-per-Block
Der Pointer zum Datenbereich des Filepuffers wird auf den Zielbereich gestellt. SET MARK liest jetzt direkt in den Zielbereich hinein. Von jetzt an müssen bei einem Fehler die Pointer vor dem Exit wieder zurückgesetzt werden!

$E275 Loop für Stufe 2: SET MARK wird aufgerufen und liest in den Zielbereich hinein.

$E27A Bleibt noch mehr als ein ganzer Block übrig, Sprung zu $E2A7.

$E29A Die Pointer werden auf den Datenbereich des Filepuffers zurückgesetzt ($E3A3). Bleiben noch einzelne Bytes übrig, Sprung zu $E24B, ansonsten Sprung zu $E2EF.

$E2A7 Stufe 3: Blockbestimmung durch den Indexblock ohne SET MARK
Wird ein neuer Indexblock benötigt, muß mit SET MARK weitergearbeitet werden: Sprung zu $E275.
Mit der Position wird der Index im Indexblock bestimmt und die Nummer des nächsten Blocks gesetzt. Ist der entsprechende Datenblock nicht besetzt (00 00 – „sparse"), wird via SET MARK weitergearbeitet: Sprung zu $E275.

$E2DA Der Datenblock wird via $D0DA gelesen, Sprung zu $E27A.

$E2E2 Fehler beim Lesen via $D0DA: Neusetzen der Filepuffer-Pointer und der anderen Parameter des Files ($E3A3). Weiter mit $E2E8.

$E2E8 Error Exit für READ: Aufruf von $E2EF und RTS mit Carry gesetzt => $D078.

$E2EF-$E305 Gemeinsamer Exit für READ und WRITE
Die Anzahl der transferierten Bytes wird im Parameterblock des „user"-Programms gespeichert. Exit via SET MARK (für den Fall, daß READ oder WRITE genau auf einer Blockgrenze endet: Der nächste Block wird gelesen, bei WRITE wird der letzte Block zurückgeschrieben. Zweiter Grund: nach READ Stufe 2 und 3 stimmt der Inhalt des Filepuffers nicht mehr. Die „Position jetzt" im FCB ist unverändert, die „Position gewünscht" steht auf der tatsächlichen Position: SET MARK bringt den Filepuffer auf den neuesten Stand.
RTS zu $D078 oder zur jeweiligen Error-Routine.

$E306-$E32F Justiert den Ziel/Quellpointer „rückwärts". Das niederwertige Byte des Zählers für den Transfer wird abgezogen, damit der Transfer in der folgenden Routine mit dem Index-Register = 0 endet.

$E315 Falls NEWLINE aktiv ist, werden die MASK und der CHAR kopiert, das Carry wird gesetzt und bleibt es während des gesamten READ-Transfers!

$E328 Der Pointer zum Filepuffer wird während des Transfer-Loops verändert, deshalb wird eine Kopie davon benutzt.
RTS zur aufrufenden Routine mit dem niederwertigen Byte der Transfer-Anzahl in X und Carry gesetzt, wenn NEWLINE aktiv ist. Wenn von WRITE aufgerufen, wird das Carry (und NEWLINE) ignoriert.

$E330-$E3A2 **READ MOVER LOOP**
Transferiert Bytes aus dem Filepuffer in den Zielbereich.
Aufruf mit dem niederwertigen Byte des Transferzählers in X.
Ist das Carry gesetzt, wird auf NEWLINE getestet.
Der Zielpointer wird am Ende der Routine um die Anzahl der transferierten Bytes erhöht.
RTS zur aufrufenden Routine bei einer von folgenden Bedingungen:
Transferzähler vollständig heruntergezählt oder NEWLINE entdeckt: V-Flag gesetzt.
Momentaner Filepuffer zu Ende: RTS mit V-Flag gelöscht.
Das Carry bleibt dabei immer unverändert!

$E3A3-$E3B0 Neusetzen des Filepuffers nach READ, Stufe 2 und 3:
Die Zielpointer für den Byte-per-Byte Transfer ($4E/$4F) werden auf die momentane Position gesetzt (falls noch einzelne Bytes zu transferieren sind). Sprung zu $D3E0, dem zweiten Teil von Set File Data und RTS zu READ von da aus.

$E3B1-$E402 READ für Directory-Files

$E403-$E40C Holt die Ziel-Startadresse aus dem Parameterblock des „user"-Programms und speichert sie auf der Zeropage. Weiter mit $E40E.

$E40E-$E414 Holt den Storage Type aus dem FCB-Eintrag. RTS zur aufrufenden Routine.

$E415-$E441 Holt den momentanen EOF aus dem FCB-Eintrag und berechnet „Position jetzt" + Request Count; d.h. Vergleich, ob die gewünschte READ/WRITE-Operation den momentanen EOF überschreiten wird.

$E433 Einsprung für den Error Exit von WRITE. Hier enthalten die Speicherstellen, mit denen der EOF verglichen wird, die „Position jetzt".
Vergleich und RTS zur aufrufenden Routine mit Carry gesetzt, wenn der EOF kleiner ist.

$E442-$E473 Teil des Error Exit von WRITE
Der EOF und die „Position jetzt" werden im FCB-Eintrag neu gesetzt.
Die „Position jetzt" wird gegen den EOF verglichen ($E433). Weiter mit $E45E.

$E45E Wenn von $E483 (WRITE: normaler Exit) aufgerufen: der EOF wird neu gesetzt, wenn „Position jetzt" + Request Count den alten EOF überschreiten.
Wenn ein Teil des Error Exit ($E442): Der EOF wird neu gesetzt, wenn die „Position jetzt" den alten EOF überschreitet.
RTS zur aufrufenden Routine.

$E474-$E47B Setzt das X-Reg auf den FCB-Index + 2, um drei Bytes zu kopieren. Nur von $E442 aus aufgerufen. RTS zu dieser Routine.

$E47C-$E51E **$CB – WRITE COMMAND**
Die Logik von WRITE ist bereits in „Logische Abläufe" beschrieben.

$E47C Der Request Count wird aus dem Parameterblock des „user"-Programms geholt und die Zähler für den Transfer-Loop gesetzt ($E3E8), „Position jetzt" und EOF werden aus dem FCB-Eintrag kopiert, „Position jetzt" + Request Count werden berechnet ($E415), der EOF wird neu gesetzt, wenn das Ergebnis den alten EOF überschreitet ($E45E).

$E486 Wenn der File READ ONLY ist, Sprung zu $E4CF mit ERR#4E: FILE ACCESS ERROR.

$E48F Wenn dies das erste WRITE seit dem OPEN des Files ist, wird „drive ready" und WRITE PROTECTED des Volumes getestet ($E64E). Bei Fehler Sprung zu $E4CF.

$E494 Ist der Request Count = 0, Sprung zu $E2EF (WRITE beendet).

$E49F Die Startadresse im Speicher wird auf die Zeropage kopiert ($E403).

$E49F Hat der File nicht den Type 1/2/3, folgt ERR#4E: FILE ACCESS ERROR und Sprung zu $E4CF, d.h. auf Directories kann nie mit WRITE geschrieben werden.

$E4A6 Start des Transfer Loop. SET MARK holt den gewünschten Block in den Speicher bzw. stellt einen leeren Filepuffer und setzt die entsprechenden Erweiterungs-Bits in STATUS. Wird keine Erweiterung und auch kein neuer Block benötigt wird, erfolgt Sprung zu $E514.

$E4B2 Die Erweiterungsbits in STATUS bestimmen auch gleichzeitig die Anzahl der benötigten Blocks (max. 3 für Erweiterung von Type 1 auf Type 3: 1 Master, 1(!) Index und 1 neuer Datenblock). Durch Aufruf von $DB66 wird festgestellt, ob auf dem Volume noch genügend Platz vorhanden ist. Wenn nicht, Sprung zu $E4CF.

$E4C3 Wenn eine Erweiterung auf Type 3 benötigt wird, Aufruf von $E58F, danach Sprung zu $E4E3. Wird keine Erweiterung auf Type 3 benötigt, Sprung zu $E4D7.

$E4CF Error Exit. Der EOF wird korrigiert ($E442), Sprung zu $E2E8, dem Error Exit von READ. RTS zu $D078 von da aus nach Setzen des Transfer Count und Aufruf von SET MARK.

$E4D7 Test, ob Erweiterung auf Type 2 bzw. ein neuer Indexblock benötigt wird. Wenn nicht, muß der File bereits mindestens den Type 2 haben, weil als nächstes ein neuer Datenblock belegt wird! Wird eine Erweiterung auf Type 2 bzw. ein neuer Indexblock (Type 3) benötigt, Aufruf von $E5DA.

$E4E3 Die notwendigen Erweiterungen des Storage Type sind erledigt. Ein neuer Datenblock wird belegt ($E62E) und in STATUS wird Bit 7 („Indexblock altered") gesetzt. Die Nummer des neuen Datenblocks wird im Indexblock und als „Datenblock momentan im Speicher" im FCB eingetragen. Die Expansion-Bits in STATUS werden gelöscht.

$E514 Der Datenblock, auf den geschrieben werden soll, ist im Speicher. Die Pointer werden justiert ($E306), die Daten werden Byte-per-Byte in den Datenteil des Filepuffers kopiert ($E51F). Wird dabei das Blockende erreicht und ist der Transfer Request noch nicht vollständig herunterge-

zählt, Loop zu $E4A6. Der dort stattfindende Aufruf von SET MARK schreibt diesen Block auf das Volume und lädt den nächsten Block bzw. setzt die entsprechenden Erweiterungsbits im STATUS des Files.

$E51C WRITE Exit: Sprung zu $E2EF, RTS zu $D078 von da aus.

$E51F-$E58E Kopiert Byte-per-Byte von der Quelladresse in den Datenteil des Filepuffers. Aufruf mit dem niederwertigen Byte des Transfer Request in X und „rückwärts" justiertem Quellpointer. Im Gegensatz zu READ wird NEWLINE ignoriert.

$E576 Am Ende der Routine werden folgende Flags gesetzt:
STATUS-Bit 6: „Data block altered";
STATUS-Bit 4: Test auf „drive ready" bereits ausgeführt (s. $E48F);
FCB-Byte $1C: „File altered", bei FLUSH muß der Eintrag im Directory immer auf den neuesten Stand gebracht werden.
RTS zur aufrufenden Routine (nur WRITE) mit entsprechend gesetztem V-Flag: Gesetzt, wenn der Request Count auf Null heruntergezählt wurde (= Ende von WRITE); gelöscht, wenn noch weitere Daten zu schreiben sind (Ende des momentanen Blocks erreicht).

$E58F Erweiterung auf Storage Type 3
Der Storage Type des Files wird um eins heraufgesetzt ($E5E7). Ist danach noch nicht Type 3 erreicht, wird $E5E7 noch einmal aufgerufen. Weiter mit $E5A0.

$E5A0-$E5D9 Erstellung eines neuen/nächsten Indexblocks
Der nächste freie Block (= nächster Indexblock) wird belegt ($E62E), die Blocknummer wird im Masterblock eingetragen, der sich momentan im Indexteil des Filepuffers befindet sowie als „Indexblock momentan im Speicher" im FCB. Der Masterblock wird zurückgeschrieben ($E081).

$E5C8 Der Indexteil des Filepuffers wird gelöscht.
RTS zur aufrufenden Routine.

$E5DA-$E5E6 Erweiterung von Type 1 => 2 oder neuer Indexblock (Type 3) benötigt.
Der momentane Storage Type des Files wird getestet, wenn der File den Type 1 hat, Sprung zu $E5E7.
Ansonsten wird der Masterblock in den Indexbereich des Filepuffers gelesen ($E02E) und Sprung zu $E5A0.

$E5E7-$E62D Erweiterung Type 1 => 2 / Type 2 => 3
Der nächste freie Block wird belegt ($E62E), die Blocknummer wird als „erster Block im File" und als „Indexblock momentan im Speicher" im FCB eingetragen. Die ehemalige Nummer des „ersten Blocks im File" aus dem FCB wird an erster Stelle im neuen Block (von SET MARK im Filepuffer gelöscht) eingetragen, der Block wird zurückgeschrieben ($E081). Der Block verbleibt im Indexbereich des Filepuffers.

$E61C Der Storage Type des Files wird um eins erhöht (FCB-Byte $07). Zusätzlich wird in STATUS Bit 3 gesetzt (s. „Spezielles"). RTS zur aufrufenden Routine.

$E62E-$E64D Belegen des nächsten freien Blocks auf dem Volume für WRITE. Der nächste freie Block wird gesucht und belegt (ALLOCATE NEXT BLOCK: $DCA9). In STATUS wird Bit 4 gesetzt (Test auf „drive ready" bei weiteren WRITEs auf diesen File nicht mehr nötig), die Anzahl der vom File belegten Blocks (FCB-Bytes $18/$19) wird um eins erhöht. RTS zur aufrufenden Routine.

$E64E-$E676 Test auf „drive ready" und WRITE PROTECTED
Wenn dieser Test seit dem OPEN des Files schon einmal durchgeführt wurde (mindestens ein Bit im Bereich Bit 4-7 von STATUS ist dann gesetzt), RTS. Ansonsten wird die Unitnummer aus dem FCB-Eintrag geholt.

$E65B Einsprung für CREATE und DESTROY mit Unitnummer in A.
Das Unit wird gesetzt, die Nummer des zuletzt benutzten Blocks wird zwischengespeichert und die RWTB des Units mit dem Befehl $00 via $D0DA aufgerufen. Die RWTB überprüft WRITE PROTECTED und „Motor on". Da der „Motor on"-Test prüft, ob sich die Daten am Lesekopf des Laufwerks ändern, wird hierbei entdeckt, ob eine Diskette eingelegt (+ Klappe geschlossen) ist, nicht aber, ob diese Diskette korrekt formatiert ist. Die RAMdisk-RWTB liefert auf den $00-Befehl immer ein „drive ready" zurück.
Die Nummer des zuletzt benutzten Blocks wird zurückgespeichert. RTS mit entsprechendem Carry (gesetzt = Fehler) zur aufrufenden Routine.

$E677-$E6E4 **$CC – CLOSE COMMAND**
Vorher wurde nur SET TIME ausgeführt.
Die Reference Number wird aus dem Parameterblock des „user"-Programms geholt. Ist sie <> 0, Sprung zu $E6B2.

$E682 CLOSE ALL
Angefangen vom ersten Eintrag wird der gesamte FCB abgesucht. Wird ein Eintrag gefunden, vergleicht CLOSE das Byte $1B des Eintrags mit LEVEL ($BF94). Ist der „Level" des momentanen behandelten Eintrags kleiner, wird dieser Eintrag übersprungen (s. „Spezielles").

$E693 Mit aus dem FCB-Eintrag geladener Reference Number wird zuerst FLUSH SPECIFIC ($E714), danach CLOSE SPECIFIC ($E6B7) aufgerufen. Tritt dabei ein Fehler auf, wird die Fehlernummer zwischengespeichert und nach dem CLOSE des letzten Files wieder geladen. RTS => $D078.

$E6B2 CLOSE SPECIFIC
FLUSH SPECIFIC ($E71C) wird aufgerufen.

$E6B7 Einsprung für den Loop von CLOSE ALL.

Der Filepuffer wird in der SBM wieder freigegeben ($EE34), der FCB-Eintrag durch Löschen der Reference Number ungültig gemacht. Der VCB-Eintrag für das Volume, auf dem der soeben geschlossene File steht, wird gesucht ($DA69). In diesem VCB-Eintrag wird die Anzahl der offenen Files (Bytes $1E/$1F) um eins heruntergesetzt. Bleibt dabei kein offener File auf dem Volume mehr übrig, wird auch das „OPEN File(s)"-Flag (Byte $11) im VCB-Eintrag gelöscht.
RTS entweder zu $D078 oder zum aufrufenden Loop von CLOSE ALL.

$E6E5 „Verlängerung" für Fehler bei CLOSE: Sprung zu $E7ED.

$E6E8-$E7EC **$CD – FLUSH COMMAND**
Vorher wurde nur SET TIME ausgeführt.
Die Reference Number wird aus dem Parameterblock des „user"-Programms geholt. Ist sie <> 0, Sprung zu FLUSH SPECIFIC ($E71C).

$E6F3 FLUSH ALL Loop
Der gesamte FCB wird der Reihe nach abgearbeitet, für jeden (besetzten) Eintrag wird FLUSH SPECIFIC ($E714) aufgerufen. Tritt dabei ein Fehler auf, wird die Fehlernummer zwischengespeichert und nach der Bearbeitung des letzten FCB-Eintrags wieder geladen.

$$E709 Eine eventuell zwischengespeicherte Fehlernummer wird geladen, das Carry wird entsprechend gesetzt. RTS zu $D078, wenn dies das Ende des Loops ist (s. $E726).

$E711 „Verlängerung" für Fehler bei FLUSH: Sprung zu $E7ED.

$E714 FLUSH SPECIFIC – Aufruf durch den Loop von CLOSE/FLUSH ALL.
Aufruf des zweiten Teils von „Set File Data" ($D3C0), Sprung zu $E726.

$E71C Einsprung für den FLUSH eines einzigen Files: Aufruf von „Set File Data" ($D3C5), also mit Verifizierung, ob der betreffende File im FCB existiert und Berechnung des FCB-Index.

$E726 Wenn der File READ ONLY ist, RTS via $E709. Ist das Flag „File altered" im FCB-Eintrag gesetzt, Sprung zu $E739.
Sind in STATUS die Bits 6 und 4 beide gelöscht (kein geänderter Datenblock im Speicher, kein WRITE auf diesen File hat stattgefunden), folgt ebenfalls RTS via $E709.

$E739 Je nach gesetzten Bits in STATUS werden Daten/Indexblock zurückgeschrieben.

$E751 Ein Update des File-Eintrags in der Directory wird vorbereitet. Um schlußendlich mit ENTRY & UPDATE, also dem Überschreiben des gesamten(!) Eintrags arbeiten zu können, müssen dazu zuerst einige Informationen zusammengesucht werden:
Unitnummer, die Nummer des Directory-Blocks, in dem der File-Eintrag steht, sowie Eintragsnummer und Nummer des Key Block der Directory werden aus dem FCB-Eintrag kopiert.

$E765 Um den Eintrag innerhalb des Directory-Blocks zu lokalisieren, wird die Eintragslänge gebraucht: der Key Block der Directory wird gelesen ($DDE1), die „Länge eines Eintrags" aus dem gelesenen Key Block geholt ($D8A0).

$E777 Wenn der File-Eintrag nicht im Key Block der Directory selber steht, wird jetzt der entsprechende Block gelesen ($DDE1).

$E78E Der Eintrag wird via Eintragsnummer und Eintragslänge bestimmt ($D692) und aus dem Directory-Block herauskopiert ($D79D).

$E794 Die Anzahl der vom File belegten Blocks, der EOF und „erster Block im File" sowie Storage Type werden mit den neuen Daten aus dem FCB-Eintrag überschrieben.

$E7CE Jetzt liegt ein vollständiges File Info im Variablenbereich vor. ENTRY & UPDATE ($D6AB) lokalisiert den alten Eintrag innerhalb des Directory-Blocks erneut, überschreibt ihn vollständig mit dem hier vorbereiteten Datensatz und arbeitet sich danach bis zur Volume Directory hoch.

$E7D3 Das Flag „File altered" (Byte $1C im FCB-Eintrag) wird zurückgesetzt.

$E7DE Wenn die VBM dieses Units momentan noch im Speicher steht (und nicht bereits beim FLUSH eines anderen Files auf diesem Unit oder beim Lesen der VBM eines anderen Units zurückgeschrieben wurde), wird das jetzt nötigenfalls nachgeholt ($DD86).
RTS entweder zu $D078 (FLUSH eines bestimmten Files), zum Loop von FLUSH ALL oder zur aufrufenden Routine (CLOSE/SET EOF).

$E7ED-$E7FB Error handler für CLOSE und FLUSH
Wenn das Argument des Command = 0 ist, d.h. das Programm befindet sich gerade in einem „ALL"-Loop, wird die Fehlernummer gespeichert und das Carry gelöscht. RTS zur aufrufenden Routine. Folge: Bei einem „ALL"-Loop werden auf jeden Fall sämtliche Files bearbeitet, treten dabei aber mehrere Fehler auf, wird nur der letzte Fehler ordnungsgemäß gemeldet.
Ist das Argument des Command <> 0, bleibt das Carry vor dem RTS zur aufrufenden Routine unverändert (gesetzt).

$E7FC-$E802 GET STATUS
Aufruf mit gesetztem FCB-Index. Byte $08 des FCB-Eintrags wird geladen. RTS zur aufrufenden Routine.

$E803-$E806 Error Exit für SET EOF. ERR#4E: FILE ACCESS ERROR. RTS => $D078.

$E807-$E94A **$D0 – SET EOF COMMAND**
Vorher wurde „Set File Data" ($D3C5) ausgeführt.
Der Storage Type des Files wird aus dem FCB geholt ($E40E). Hat der File nicht den Type 1/2/3, Sprung zu $E803, d.h. SET EOF ist in Directories nicht möglich.
Ist der File READ ONLY gesetzt, ebenfalls Sprung zu $E803.

$E81C „Drive ready" wird getestet ($E64E). Bei Fehler (WPROT) => $E803.

$E821 Der momentane EOF des Files wird aus dem FCB, der gewünschte EOF aus dem Parameterblock des „user"-Programms kopiert, die beiden Werte werden miteinander verglichen. Ist der gewünschte EOF kleiner als der momentane, Sprung $E863; sonst weiter mit $E84E.

$E84E-$E860 Der neue EOF wird im FCB gesetzt. RTS zu $D078 via $EC50, wo das Flag „File altered" (Byte $1C) im FCB-Eintrag gesetzt wird.

$E863 Der neue EOF ist kleiner als der alte. FLUSH ($E71C) wird ausgeführt, um eventuell noch verbliebene Blocks aus dem Filepuffer zu schreiben. FLUSH kehrt zurück via SET MARK und SET MARK hat den Daten-Pointer im Filepuffer (unerwünschterweise) auf die momentane Position (Speicherseite) gesetzt. Der Pointer wird auf den Anfang des Datenteils zurückgesetzt.

$E872 Die „Position jetzt" wird mit dem gewünschten EOF verglichen. Ist die „Position jetzt" größer als der neue EOF, muß die Position korrigiert werden. Sie wird dann auf den neuen EOF gesetzt.

$E898 Die Parameter für REMOVE BLOCKS werden gesetzt: aus dem neuen EOF ergibt sich die Anzahl der Blocks, die der File hinterher haben soll, d.h. die Untergrenze, bei der REMOVE BLOCKS aufhört, weitere Blocks aus dem File zu entfernen. Ist der neue EOF genau auf einer Blockgrenze, also auf Position 00 in einem Block, kann dieser Block ebenfalls mit entfernt werden: die Untergrenze wird um einen weiteren Block heruntergesetzt. Ergibt sich dabei ein Wert kleiner Null, werden alle Untergrenzen auf Null gesetzt. Es ist also mit SET EOF möglich, den Inhalt eines Files vollständig zu löschen.

$E8D8 Die Nummer des „ersten Blocks im File" wird kopiert (sie ändert sich, wenn der Storage Type durch REMOVE BLOCKS herabgesetzt wird), ein Zähler für die Anzahl der entfernten Blocks wird mit Null initialisiert und REMOVE BLOCKS ($EC62) aufgerufen.

$E8F5 Die Nummer des „ersten Blocks im File" wird wieder in den FCB übertragen. Von der Anzahl der durch den File belegten Blocks wird die Anzahl der entfernten Blocks abgezogen und der FCB-Eintrag auf den neuen Stand gebracht.

$E910 Der (eventuell ebenfalls veränderte) Storage Type wird in den FCB zurückgeschrieben.

$E91D Eventuell gesetzte Expansion-Bits in STATUS (woher?) werden gelöscht ($DFA2), die Anzahl der verfügbaren Blocks auf dem Volume wird um die Anzahl der entfernten Blocks erhöht.

$E923 Die „Position jetzt" wird aus dem FCB als „Position gewünscht" kopiert, „umgedreht" und als „Position jetzt" gespeichert. Danach wird SET MARK aufgerufen und liest aufgrund der verdrehten Position den „momentanen" Block neu in den Filepuffer.
Grund: REMOVE BLOCKS arbeitet nicht im Filepuffer, d.h. wird der Storage Type herabgesetzt oder ein Teil des „momentanen" Blocks gelöscht, enthält der Filepuffer ungültige Daten.

$E945 Der neue EOF wird im FCB eingetragen ($E84E). RTS => $D078.

$E94B-$E95C **$D1 – GET EOF COMMAND**
Vorher wurde „Set File Data" ($D3C5) ausgeführt.
Der EOF wird aus dem FCB-Eintrag in den Parameterblock des „user"-Programms kopiert. RTS => $D078.

$E95D-$E96E **$C9 – NEWLINE COMMAND**
Vorher wurde „Set File Data" ($D3C5) ausgeführt.
Aus dem Parameterblock des „user"-Programms werden NEWLINE MASK auf Byte $1F und NEWLINE CHAR auf Byte $0A des FCB-Eintrags kopiert. RTS => $D078.

$E96F-$E9D3 **$C4 – GET FILE INFO COMMAND**
Vorher wurde Copy & Verify Pathname ($D27F) ausgeführt.
Der File wird gesucht, der File-Eintrag geladen (GET FILE: $D798).
Tritt dabei kein Fehler auf, Sprung zu $E9B6.
Ist der Fehler ERR#40 (INVALID PATHNAME), endete FIND FILE ENTRY in einer Volume Directory. Weiter mit $E979. Andere Fehler sind nicht zulässig (RTS mit Carry gesetzt).

$E979 GET F'INFO für Volume Directories
Der von GET FILE „getürkte" Storage Type wird wieder auf $0F ($F0) gesetzt. Um die Anzahl der freien Blocks zu erhalten, wird „benötigte Blocks = 00" gesetzt und „Space on Volume" ($DB71) aufgerufen. Bei diesem Aufruf wird die VBM immer gelesen und die Berechnung durchgeführt, auch wenn der VCB-Eintrag für das Volume bereits die Anzahl der freien Blocks enthält. Dabei auftretende Fehler (nicht nur VOLUME FULL) werden überhaupt nicht berücksichtigt(!).

$E99B Die Anzahl der belegten Blocks ergibt sich aus der Gesamtzahl der Blocks für das Volume gemäß dem VCB-Eintrag minus der zurückgelieferten Anzahl der freien Blocks. Weiter mit $E9B6.

$E9B6 GET F'INFO für Subdirectories/Files
Der Storage Type des gefundenen Files wird heruntergeshiftet.
Mit Hilfe einer Adress-Übersetzungstabelle auf $EFCC wird die durch GET FILE gewonnene Information in den Parameterblock des „user"-Programms kopiert. RTS => $D078.

$E9D4-$EA08 **$C3 – SET FILE INFO COMMAND**
Der File wird gesucht (GET FILE: $D798). SET F'INFO für Volume Directories ist nicht möglich, deshalb führt hier jeder Fehler (auch ERR#40) zum Abbruch (RTS zu $D078 mit gesetztem Carry).

$E9D9 Das „Backup-Bit" (ACCESS - Bit 5 im File-Eintrag) wird nur beibehalten, wenn es bereits vorher gesetzt war und BACKUPBIT auf der Global Page ($BF95) nicht gesetzt ist (s. „Spezielles").

$E9E6 Das neue FILE INFO wird mit Hilfe einer Adress-Übersetzungstabelle auf $EFCC kopiert, d.h. der durch GET FILE gewonnene Datensatz wird teilweise überschrieben. Die Adress-Übersetzungstabelle ist dieselbe wie für GET F'INFO, nur daß hier das „Null Field" ausgelassen wird.

$E9F7 Erst jetzt wird der (neue) Storage Type des Files geprüft: ist er nicht im Bereich 0(!)/1/2/3, folgt ERR#4E: FILE ACCESS ERROR und RTS zu $D078.

$E9FF War MODIFICATION DATE & TIME im Parameterblock des „user"-Programms gesetzt, folgt ENTRY & UPDATE ohne das vorherige Lesen der System Time ($D6BB), ansonsten wird die System Time in ENTRY & UPDATE gelesen ($D6AB). RTS zu $D078 von ENTRY & UPDATE aus.

$EA0B-$EB4F **$C2 – RENAME COMMAND**
Vorher wurde Copy & Verify Pathname ($D27F) ausgeführt.
Der File wird gesucht (FIND FILE ENTRY: $D7AB). Wird er ohne Fehler gefunden, Sprung zu $EA4D. Bei ERR#40 hat FIND FILE ENTRY eine Volume Directory gefunden, weiter mit $EA14. Andere Fehler sind nicht erlaubt: RTS zu $D078 mit gesetztem Carry.

$EA14 RENAME Volumes: Der Pointer zum neuen Namen wird aus dem Parameterblock des „user"-Programms geholt, verifiziert und auf $F100... kopiert ($EB35). Der neue Path darf auch nur aus einem Namen bestehen, ansonsten RTS mit ERR#40: INVALID PATHNAME.

$EA22 Wenn auf diesem Volume ein offener File existiert (VCB-Byte $11, Bit 7 gesetzt), RTS mit ERR#50: FILE IS OPEN.

$EA2E Der neue Name wird um den Storage Type $0F ($F0) bereichert. FIND FILE ENTRY hat den Key Block der Volume Directory bereits geladen. Der Name wird in den Key Block kopiert und der Block wird zurückgeschrieben ($EB26).

$EA3C Der neue Name wird in den VCB-Eintrag kopiert und überschreibt damit den alten Namen für das Volume. RTS => $D078.

$E4AD RENAME für Subdirectories/Files:
Der alte Name wurde von Copy & Verify Pathname auf $F100... kopiert. Ist der Name unvollständig, gehört noch das System Prefix dazu, das auf $F1C0..$F1FF steht. Der Aufruf von $EB43 setzt den entsprechenden Startindex.

$EA50 Der alte Pathname wird in den Directory Space ($F600-) kopiert.

$EA5C Der Pointer zum neuen Namen wird aus dem Parameterblock des „user"-Programms geholt, der Name wird verifiziert und kopiert ($EB35). Der Startindex wird erneut geholt ($EB43) und die beiden Namen werden miteinander verglichen. Sind beide Namen vollständig gleich, folgt RTS zu $D078 ohne Fehler.

$EA81 Es wurde ein Unterschied gefunden: Tritt dieser Unterschied nicht im letzten Namen des neuen Path auf, RTS mit ERR#40: INVALID PATHNAME.

$EA91 Enthält der alte Path noch weitere Namen, ebenfalls RTS mit ERR#40.

$EAA5 Mit dem neuen Namen wird FIND FILE ENTRY ($D7AB) aufgerufen. Ergibt sich dabei kein Fehler (d.h. der File wurde gefunden), RTS zu $D078 mit ERR#47: DUPLICATE FILENAME.

$EAAE Ist ein anderer Fehler als ERR#46 aufgetreten, erfolgt natürlich ebenfalls ein Abbruch und RTS zu $D078 mit entsprechender Fehlernummer.

$EAB2 Mit Copy & Verify Pathname wird der ursprüngliche Name wieder geladen. Der File wird gesucht (GET FILE: $D798). Der FCB wird nach diesem File abgesucht ($E1A9). Ist der File (WRITE erlaubt) im FCB, folgt ERR#50: FILE OPEN.

$EAC1 Wenn RENAME nicht erlaubt ist (ACCESS aus dem File-Eintrag, durch GET FILE geladen), folgt ERR#4E: FILE ACCESS ERROR.

$EACC Hat der File nicht den Storage Type 1/2/3 oder Subdirectory ($0D), folgt ERR#4A: INCOMPATIBLE FILE FORMAT.

$EADD Der neue Path wird wieder geholt und verifiziert ($EB35). Der beim vorherigen Vergleich beider Namen gespeicherte Index ist der Start des neuen Namens.

$EAE5 Der neue Name wird kopiert und um den Storage Type bereichert. Damit ist wieder ein vollständiger File-Eintrag vorbereitet.

$EB03 Wenn der zu RENAMEnde File keine Subdirectory ist, Sprung zu $EB23. Ansonsten muß der Name im Key Block der Subdirectory ebenfalls geändert werden: Die Blocknummer wird gesetzt, der Block wird in den Directory Space ($F600-$F7FF) gelesen. Der Name wird um den Storage Type $0E ($E0) bereichert. Der Name wird in den Key Block kopiert, der Block wird zurückgeschrieben ($EB26).

$EB23 Sprung zu ENTRY & UPDATE ($D6BB). MODIFICATION DATE & TIME werden nicht neu gesetzt. Der alte Eintrag wird lokalisiert und mit dem neuen Namen überschrieben. Danach arbeitet sich ENTRY & UPDATE unnötigerweise bis zur Volume Directory hinauf. RTS zu $D078 von dort aus.

$EB26-$EB34 Kopiert einen Namen aus dem Path von $F100.. in den Directory Space (Start $F604). Aufruf mit gesetztem Anfangsindex innerhalb des Path. Der Block wird zurückgeschrieben ($DDDD). RTS zur aufrufenden Routine von dort aus. Nur innerhalb von RENAME benutzt.

$EB35-$EB42 Holt den Pointer zum neuen (Path)namen aus dem Parameterblock des „user"-Programms und springt zu Copy & Verify Pathname (zweiter Einsprung, Pointer bereits gesetzt). Der neue Name wird verifiziert und nach $F100.. kopiert. RTS zur aufrufenden Routine von dort aus. Nur innerhalb von RENAME benutzt.

$EB43-$EB4F Lädt den Startindex für den momentanen Path. Wenn der Name vollständig war (FULL NAME FLAG: $F07C), wird der Start auf den Anfang des neuen Namens ($00), ansonsten auf das System Prefix ($BF9A) gesetzt. RTS zur aufrufenden Routine (nur innerhalb von RENAME benutzt).

$EB50-$EC4A **$C1 – DESTROY COMMAND**
Vorher wurde Copy & Verify Pathname ($D27F) ausgeführt.

Der File wird gesucht (GET FILE: $D798), der FCB wird nach diesem File abgesucht ($E1A9). Wurde dieser File gefunden (auch READ ONLY), folgt ERR#50: FILE IS OPEN.

$EB61 Die Anzahl „Blocks benötigt" wird auf Null gesetzt und „Space on Volume" ($DB66) aufgerufen, um die Gesamtzahl der freien Blocks auf dem Volume zu erhalten.

$EB73 Ist im von GET FILE geladenen ACCESS des File-Eintrags DESTROY nicht erlaubt, folgt RTS mit ERR#4E zu $D078 via SET SYSERR ($BF09).

$EB82 „Drive ready" und WRITE PROTECTED werden auf dem Volume getestet ($E65B).
Die Nummer des „ersten Blocks im File" und der Storage Type werden kopiert, sie ändern sich während REMOVE BLOCKS.

$EB98 Ist der Storage Type größer als 3, Sprung zu $EBF8.

$EBA0 DESTROY für Files mit Storage Type 1/2/3.
Die Anzahl der bis jetzt freigegebenen Blocks und die Position, bei der REMOVE BLOCKS aufhören soll, weitere Blocks zu entfernen, werden auf Null gesetzt. Die Untergrenze für das Löschen innerhalb des letzten Blocks wird auf $200 gesetzt, ein Löschen von Daten innerhalb dieses letzten Blocks ist im Gegensatz zu SET EOF nicht nötig, weil der Block ebenfalls vollständig entfernt wird.

$EBB2 REMOVE BLOCKS ($EC62) wird aufgerufen. Das Ergebnis ist identisch mit einem SET EOF auf die Position $0200, d.h. der File hat zum Schluß den Storage Type 1 und einen Datenblock.

$EBB7 Für Datenfiles: Dieser Datenblock (der jetzt „erste Block im File") wird auch noch entfernt ($DC27).
Für Directory-Files (s. $EC27) ist der „erste Datenblock" der Key Block, der Vorgang ist derselbe.
Der komplette Datensatz für den File-Eintrag ist immer noch geladen: der Storage Type wird auf 0 gesetzt und die Anzahl der aktiven Einträge im Datensatz für die Parent Directory um eins verringert.
Der momentan im Speicher befindliche VBM-Block wird zurückgeschrieben ($DD86), die Anzahl der verfügbaren Blocks auf dem Volume (VCB-Eintrag) wird um die Anzahl der freigegebenen Blocks erhöht ($EBDD).

$EBDA Sprung zu ENTRY & UPDATE ($D6AB). Der Parent Directory-Block wird geladen, der File-Eintrag lokalisiert und überschrieben. Dabei ändern sich nur der Storage Type, die Anzahl der durch den File belegten Blocks und MODIFICATION DATE & TIME. Witzigerweise wird MODIFICATION DATE & TIME auch im gerade gelöschten File-Eintrag neu gesetzt.

$EBDD-$EBF7 Die Anzahl der freien Blocks auf dem Volume im FCB-Eintrag wird um die Anzahl der freigegebenen Blocks erhöht, die „VBM Extent Number" (VCB-Byte $1C) wird auf Null (erster Block) zurückgesetzt. RTS zur aufrufenden Routine (DESTROY/SET EOF).

$EBF8 DESTROY für Directories bzw. Typen > 3
Wenn das File nicht den Storage Type $0D (Subdirectory) hat, folgt ERR#4E: INCOMPATIBLE FILE FORMAT und Exit via SET SYSERR.
Der erste VBM-Block des Volumes wird gelesen ($DD57).

$EC01 Der Key Block der Subdirectory wird gelesen ($DDE1). Ist die Anzahl der aktiven Einträge ungleich Null, RTS zu $D078 mit ERR#4E: FILE ACCESS ERROR via SET SYSERR ($BF09).

$EC1F Der Storage Type innerhalb des Key Blocks wird auf Null gesetzt, der Block wird zurückgeschrieben ($DDDD). Der Key Block wird (noch) nicht wieder freigegeben.

$EC27 Wenn keine Forward Reference mehr existiert (00 00), war dies der letzte Block der Subdirectory: Sprung zu $EBB7.

$EC33 Der Block, auf den die Forward Reference zeigt, wird zuerst freigegeben ($DC27) und danach gelesen ($DDE1). Loop zu $EC27.

$EC4A RTS zu $D078, für Fehler während der Disk-Operationen.

$EC4B-$EC4F Error Exit. ERR#4A: INCOMPATIBLE FILE FORMAT. Exit zu $D078 via SET SYSERR ($BF09). Benutzt, wenn der File weder den Type 1/2/3 noch den Type $0D hat.

$EC50-$EC61 Setzt „File altered" (Byte $1C, Bit 7) im FCB-Eintrag („FLUSH always"). A-Y werden zwischengespeichert. RTS zur aufrufenden Routine (SET EOF / WRITE). Hat hier von der Speicherplatzlogik her nichts zu suchen.

$EC62-$EDD6 REMOVE BLOCKS (nur für Typen 1/2/3)
Wird von SET EOF und DESTROY benutzt. SET EOF setzt eine entsprechende Untergrenze, DESTROY setzt diese Untergrenze auf $200 und läßt damit nur noch einen Block übrig, der dann innerhalb von DESTROY ebenfalls komplett entfernt wird.
Der Storage Type des Files wird je nach Größe des „Restfiles" stufenweise heruntergesetzt (Umkehrung des Erweiterungsprozesses von WRITE).
Die Bearbeitung der Blocks findet im Directory Space ($F600-$F7FF) statt, weil für DESTROY kein Filepuffer zur Verfügung steht. Folge: Bei SET EOF stimmt hinterher der Inhalt des Filepuffers nicht mehr und muß (via SET MARK) neu geladen werden.

$EC62 Verzweigung für die verschiedenen Storage Types.
Type 1: Sprung zu $ED46
Type 2: Sprung zu $ED0D
Type 3: Sprung zu $EC7A

$EC71 Nicht Type 1/2/3: SYSTEM DEATH mit dem Storage Type des Files (*16) als Fehlernummer (s. „Spezielles").

$EC7A REMOVE BLOCKS für Storage Type 3

Innerhalb des Masterblocks wird von hinten nach vorne gearbeitet, d.h. der letzte Indexblock im File wird zuerst gelöscht. Innerhalb der Indexblocks wird von der gegebenen Untergrenze bis zum Ende des Blocks gelöscht, der Prozeß verläuft hier also in die entgegengesetzte Richtung, das Ergebnis (Löschung von einer Untergrenze bis zum Ende des Files) ist natürlich dasselbe.

Im Gegensatz zu Indexblocks wird der Masterblock für je acht existierende(!) Indexblocks je einmal gelesen, d.h. 22 Mal für einen File von 16 MByte.

Ein Index wird auf den maximalen Wert ($80) im Masterblock gesetzt.

$EC7F Der „erste Block im File", also der Masterblock, wird in den Directory Space ($F600-$F7FF) geladen ($ED71).

$EC84 Hat der Zähler die Untergrenze (SET EOF: beliebig, DESTROY: 0) erreicht, Sprung zu $ECE0 mit geladenem Masterblock. Der Indexblock, auf den die Untergrenze zeigt, wird nicht gelöscht, d.h. auch für DESTROY bleibt der erste Indexblock des Files erst einmal bestehen.

$EC8C Es werden bis zu acht existierende Indexblock-Blocknummern („sparse"-Blocks bzw. Blocks, die nie belegt waren, werden übersprungen) aus dem Masterblock kopiert, der Index innerhalb des Masterblocks wird dabei jedesmal gegen die Untergrenze geprüft. Ist die Untergrenze erreicht, wird der Rest der kopierten Liste mit Nullen aufgefüllt.

$ECB4 Die Blocknummer eines Indexblocks aus der kopierten Liste wird gesetzt. Ist die Nummer gleich 00 00, ist das Ende der Liste erreicht: Sprung zu $EC7F.

Ansonsten wird der Indexblock in den Directory Space geladen ($DDE1), sämtliche Datenblocks des Indexblocks werden freigegeben ($EDA0). Sind noch nicht alle 8 Indexblocks bearbeitet, Loop zu $ECB4, ansonsten Sprung zu $EC7F. Die Indexblocks selber bleiben hier noch belegt!

$ECE0 Sprung hierher mit geladenem Masterblock, wenn die gewünschte Anzahl von Indexblocks entfernt worden ist (SET EOF: beliebig, bis auf den ersten Indexblock; DESTROY: alle bis auf den ersten Indexblock).

Die Indexblocks werden nun selber aus dem Masterblock entfernt ($EDA2), der Masterblock wird zurückgeschrieben ($DDDD).

$ECEE Wenn sämtliche Indexblocks aus dem Master entfernt werden sollen (Untergrenze = 00), Sprung zu $ED08.

Der jetzt letzte Indexblock wird bestimmt. Existiert der Block nicht (00 00), folgt RTS zur aufrufenden Routine (DESTROY/SET EOF). Ansonsten wird der Indexblock geladen ($DDE1). Sprung zu $ED12.

$ED08 Der Masterblock wird freigegeben, der Indexblock als „erster Block im File" gesetzt und der Storage Type auf 2 herabgesetzt ($ED7E). Weiter mit $ED0D.

$ED0D REMOVE BLOCKS für Storage Type 2

Der „erste Block im File" wird gelesen ($ED71).

$ED12 Einsprung, wenn der File immer noch den Type 3 hat: der jetzt letzte Indexblock des Files befindet sich im Speicher.

Ausgehend vom Startindex innerhalb des Indexblocks (+1) werden die Datenblocks freigegeben, auf die der Indexblock zeigt ($EDA2). Der Block, auf den die Untergrenze zeigt (SET EOF: beliebig, DESTROY: 0), bleibt dabei erhalten.

$ED22 Wenn die Untergrenze = 0 war, Sprung zu $ED3C.

$ED27 Der jetzt letzte Datenblock des Files wird bestimmt. Ist dieser Datenblock nicht im Indexblock enthalten, folgt RTS zur aufrufenden Routine (DESTROY/SET MARK). Ansonsten wird der Datenblock geladen ($DDE1). Sprung zu $ED4B.

$ED3C Die Untergrenze im Indexblock ist auf 00 gesetzt. Hat der File noch einen Masterblock (die Untergrenze im Masterblock ist <> 0), folgt Sprung zu $ED27, der erste Datenblock (Index 00) wird bestimmt und gelesen, wenn er existiert.
Existiert dagegen kein Masterblock (mehr), ist dies der einzige Indexblock, der bis auf den ersten Datenblock gelöscht wurde. $ED7E setzt die Nummer des „ersten Blocks im File" auf den übriggebliebenen Datenblock, gibt den Indexblock selber frei und setzt den Storage Type des Files auf 1 herunter. Weiter mit $ED46.

$ED46 REMOVE BLOCKS für Files mit Storage Type 1
Der „erste Block im File" wird gelesen.

$ED4B Einsprung, wenn sich der jetzt letzte Datenblock des Files (Typen 2 und 3) bereits im Speicher befindet.
Ist die Untergrenze innerhalb dieses Blocks auf $200 gesetzt (DESTROY: immer), folgt RTS zur aufrufenden Routine(SET EOF / DESTROY).
Ansonsten wird innerhalb des Blocks ab der Untergrenze gelöscht und der Block danach zurückgeschrieben ($DDDD). RTS zur aufrufenden Routine von dort aus.

$ED71-$ED7B Setzt die Nummer des „ersten Blocks im File" als RWTB-Parameter und springt zu $DDE1 (Lesen des Blocks in den Directory Space). RTS zur aufrufenden Routine von dort aus (nur von REMOVE BLOCKS benutzt).

$ED7E-$ED9F Der jetzige „erste Block im File" wird freigegeben ($DC27), die Nummer des „ersten Blocks im File" wird auf den ersten Block innerhalb des gerade freigegebenen Blocks gesetzt (Master=> Index, Index => Datenblock). Der Storage Type des Files wird um eins ($10) heruntergesetzt. RTS zur aufrufenden Routine (nur REMOVE BLOCKS).

$EDA0-$EDD6 Freigabe aller Blocks, auf die ein Master/Indexblock zeigt.

$EDA2 Aufruf mit gesetzter Untergrenze.
Die zuletzt als RWTB-Parameter gesetzte Blocknummer wird zwischengespeichert, genauso wie die Untergrenze (Startindex).

$EDA8 Ist der Block, auf den der Index momentan zeigt, bereits 00 00, Sprung zu $EDCA.

Ansonsten wird der Block freigegeben ($DC27) und die Blocknummer innerhalb des Master/Indexblocks auf 00 00 gesetzt.

$EDCA Der Index wird neu geladen und erhöht. Loop zu $EDA8, bis das Ende des Master/Indexblocks erreicht ist.

$EDCD Die zwischengespeicherte Blocknummer wird zurückgeladen. (Das Freigeben eines Blocks kann das Lesen/Zurückschreiben eines VBM-Blocks nötig machen.) Die zuletzt gesetzte Blocknummer vor dem Aufruf von $EDA0 wird noch gebraucht, um den Master/Indexblock selber freizugeben. RTS zur aufrufenden Routine (nur REMOVE BLOCKS).

$EDD7-$EE25 Holt die Startadresse des (zukünftigen) Filepuffers aus dem Parameterblock des „user"-Programms, prüft gegen die SBM und belegt den Puffer in der SBM.

$EDD7 Einsprung für OPEN: Pufferstart sind die Bytes $04/5 im Block.

$EDD9 Einsprung für SET BUF: Pufferstart sind die Bytes $03/4 im Block.
Ist der Puffer nicht im Bereich $08xx - $BCFF, folgt ERR#56.

$EDE4 Die Startadresse wird auf die Zeropage kopiert. Fängt der Puffer nicht auf einer Seitengrenze an, folgt ERR#56.
Der Puffer umfaßt vier Seiten. Für jede Seite wird der entsprechende Index und das entsprechende Bit berechnet ($EE5D).

$EDF5 Ist das Bit im entsprechenden Byte der SBM bereits gesetzt, d.h. die Seite bereits belegt, folgt ERR#56.

$EDFE Der Test ist erfolgreich verlaufen. Wieder wird für jede Seite einzeln der Index und das entsprechende Bit in der SBM berechnet ($EE5D), aber dieses Mal wird das entsprechende Bit in der SBM gesetzt (= Seite belegt).

$EE10 Die Reference Number des Files mal 2 ergibt den Pufferindex im FCB-Eintrag (Byte $0B), der Pufferstart wird unter demselben Index in der Puffertabelle auf der Global Page gespeichert. RTS zur aufrufenden Routine.

$EE22 Error exit. Setzt ERR#56: BAD BUFFER ADDRESS und das Carry. RTS zur aufrufenden Routine.

$EE26-$EE33 Kopiert die Startadresse des Filepuffers von der BUFFER TABLE in der Global Page in den Variablenbereich. Aufruf mit gesetztem Pufferindex. RTS zur aufrufenden Routine.

$EE34-$EE5C **DEALLOCATE FILEBUFFER**
Dieser Einsprungspunkt wird nur von CLOSE und vom Error Exit von OPEN benutzt.
Die Startadresse des Puffers wird aus der BUFFER TABLE in den Variablenbereich kopiert ($EE26). Ist kein Puffer definiert (Startadresse ist 00 00), RTS zur aufrufenden Routine.
Ansonsten wird die Startadresse des Puffers in der Global Page auf Null gesetzt.

$EE43 Einsprung für SET BUF: die Startadresse in der BUFFER TABLE der Global Page ist bereits die des neuen Puffers, die Startadresse im Variablenbereich die des alten Puffers.
Die Startadresse wird um $300 erhöht, danach wird für jede der vier Speicherseiten des Puffers einzeln der Index und das entsprechende Bit in der SBM berechnet. Das Bit wird gelöscht, d.h. die Speicherseite wieder freigegeben. RTS zur aufrufenden Routine.

$EE5D-$EE6B Berechnung des Index innerhalb der SBM und des entsprechenden Bits für eine Speicherseite. Aufruf mit der Seitennummer (höherwertiges Byte der Adresse) im Akku.
RTS zur aufrufenden Routine mit Index in Y und einem entsprechend gesetzten Bit im Akku. Benutzt eine Tabelle auf $EFC0.

$EE6C-$EE88 Prüfung eines Puffers beliebiger Länge gegen die SBM
Aufruf mit Pufferstart als Pointer in der Zeropage und gesetzter Länge.
Unterschreitet der Pufferstart $200, folgt ERR#56.
Pufferstart + Länge ergeben die Obergrenze, die ebenfalls in der Zeropage gespeichert wird. Liegt dieser Wert über $BEFF, folgt ERR#56.

$EE8A Einsprung für BLOCKREAD/WRITE, Start- und Endwert sind bereits auf der Zeropage gesetzt.
Hier wird nicht auf die Bereichsgrenzen der System Bit Map getestet, d.h. bei Quell/Zieladressen von BLOCKREAD/WRITE, die größer als $BFFF sind, ergibt sich Unsinn (s.„Spezielles").
Ausgehend vom Startwert wird für jede einzelne Seite der Index und das entsprechende Bit in der SBM berechnet ($EE5D) und gegen die SBM geprüft. Ist das Bit bereits gesetzt, folgt ERR#56.
Es handelt sich dabei nur um eine Prüfung, die Seite wird nicht in der SBM belegt. Loop zu $EE8A, bis der Endwert (Untergrenze) des Pufferbereichs wieder erreicht ist. RTS zur aufrufenden Routine.

$EE98-$EEA6 **$D2 – GET BUF COMMAND**
Vorher wurde „Set File Data" ($D3C5) ausgeführt.
Die von $D3C5 aus der BUFFER TABLE der Global Page kopierte Startadresse des Filepuffers wird in den Parameterblock des „user"-Programms kopiert. RTS zu $D078.

$EEA7-$EECE **$D3 – SET BUF COMMAND**
Vorher wurde „Set File Data" ($D3C5) ausgeführt.
Die Startadresse des neuen Filepuffers wird aus dem Parameterblock des „user"-Programms geholt.
Der neue Puffer wird gegen die SBM geprüft und belegt, die Startadresse wird in der BUFFER TABLE der System Global Page eingetragen ($EDD9).
Hat $EDD9 nicht mit Fehler abgebrochen, wird die Startadresse des alten Puffers auf die Zeropage kopiert, der alte Puffer wird in der SBM freigegeben ($EE43).

$EEBB Der Inhalt des alten Puffers wird in den neuen Puffer kopiert ($400 Bytes). RTS zu $D078.

$EECF-$EF24 **$65 – REBOOT COMMAND – Teil 1**
Die Bank 2 der Language Card wird angeschaltet.
Die Speicherstellen $0000-$0003 auf der Zeropage werden zwischengespeichert.
Der Bereich $D100-$D3FF aus der Bank 2 wird nach $1000-$12FF kopiert.
Die Speicherstellen $0000-$0003 werden zurückgespeichert.
Die Bank 1 der Language Card wird wieder angeschaltet.

$EF13 Der RESET-Vektor ($03F2-$03F4) wird auf $1000 gesetzt.

$EF22 Sprung zu $1000, dem REBOOT-Programm.

$EF25-$EFDE **TABELLEN**

$EF25 COMMAND-Bytes der MLI-Befehle (Verifizierung)

$EF45 Anzahl der Parameter für die Commands (Verifizierung)

$EF65 Startadressen der Commands im Bereich $C0-$D3

$EF8D Ein Byte pro Command im Bereich $C0-$D3. Enthält den Index in der Tabelle der Startadressen (Bits 0-4) und die Indikatoren für die vorher auszuführenden Routinen (Bit 7: $D27F, Bit 6: $D3C5, Bit 5: $BF06). Bei gesetztem Bit wird die entsprechende Routine ausgeführt.

$EFA1 Additionstabelle für das Anhängen eines Subdirectory-Blocks: Die Anzahl der durch die Subdirectory belegten Blocks wird um eins, der EOF der Subdirectory um $200 erhöht.

$EFA6-$EFB7 Teil eines Subdiretory Key Blocks. „uHUSTON" etc.

$EFB0 MLI-Version: „first", „last"

$EFB2 ACCESS, Länge eines Eintrags, Anzahl der Einträge pro DIR-Block.

$EFB8 „Getürkter" Datensatz, falls FIND FILE ENTRY in einer Volume Directory endet: Storage Type $0F (nicht benutzt), „erster Block im File" auf 00 02, 00 04 belegte Blocks und EOF = 00 08 00.

$EFC0 Bit Mask für die SBM und VBM-Belegung.

$EFC8 Adressen-Übersetzungstabelle für GET/SET FILE INFO. Bei drei Bytes innerhalb dieser Tabelle ist Bit 7 zusätzlich gesetzt und indiziert das Überspringen des „Null Field" bei der Kopieraktion.

$EFDF-$F004 „INSERT SYSTEM DISK AND RESTART" – Text für SYSTEM DEATH.
Belegt die ersten 5 Bytes im Variablenbereich mit Leerzeichen.

# 1.6. Kommentiertes Listing des MLI

## Einsprung in das MLI von der GLOBAL PAGE via JMP (nicht JSR!)

Alle MLI-Befehle enden via $D078 mit MLIEXIT ($BFA0). Ausnahmen: REBOOT, der INT-Service und SYSTEM DEATH während der Befehlsausführung (oder während des INT-Service).

```
D000-   D8          CLD             ; dient nur als BANKID
D001-   8C 9F BF    STY   $BF9F     ; YTEMP
D004-   8E 9E BF    STX   $BF9E     ; XTEMP
D007-   68          PLA             ; Returnadresse zum „user"-Programm
D008-   85 40       STA   $40
D00A-   18          CLC
D00B-   69 04       ADC   #$04      ; +4 => COMMAND und Adresse des Parm-
D00D-   8D 9C BF    STA   $BF9C       blocks werden übersprungen (3 Bytes)
D010-   68          PLA
D011-   85 41       STA   $41       ; MLIRTS in der System Global Page
D013-   69 00       ADC   #$00
D015-   8D 9D BF    STA   $BF9D
D018-   A0 00       LDY   #$00      ; SYSERR = 0. Wird bei Fehler während der
D01A-   8C 0F BF    STY   $BF0F       Befehlsausführung gesetzt und beim Exit
D01D-   C8          INY               geladen (0 = kein Fehler)
D01E-   B1 40       LDA   ($40),Y   ; COMMAND
D020-   4A          LSR             ; CMD + hinibble CMD =>
D021-   4A          LSR             ; $40=>$44 $41=>$45 $81=>$89 $82=>$8A
D022-   4A          LSR             ; $C0=>$CC $C1=>$CD ....
D023-   4A          LSR
D024-   18          CLC
D025-   71 40       ADC   ($40),Y
D027-   29 1F       AND   #$1F      ; Reduktion auf $00-$1F
D029-   AA          TAX             ; Index in der MLI-eigenen COMMAND Table
D02A-   B1 40       LDA   ($40),Y   ; COMMAND aus dem „user"-Programm
D02C-   DD 25 EF    CMP   $EF25,X   ; Überprüfung
D02F-   D0 76       BNE   $D0A7     ; ERR #1: BAD SYSTEM CALL NUMBER
D031-   C8          INY
D032-   B1 40       LDA   ($40),Y   ; Pointer zum Parmblock des „user"-Programms
D034-   48          PHA               wird auf $0040/41 kopiert
D035-   C8          INY
D036-   B1 40       LDA   ($40),Y
D038-   85 41       STA   $41
D03A-   68          PLA
D03B-   85 40       STA   $40
D03D-   A0 00       LDY   #$00
D03F-   BD 45 EF    LDA   $EF45,X   ; Tabelle mit der Anzahl der Parameter
D042-   F0 1C       BEQ   $D060     ; Sprung für SET TIME ($82): 0 Parameter
D044-   D1 40       CMP   ($40),Y   ; ansonsten Vergleich
D046-   D0 63       BNE   $D0AB     ; ERR #4: BAD SYST CALL PARAM COUNT
D048-   BD 25 EF    LDA   $EF25,X   ; reload CMD (diesmal aus der MLI-Table)
D04B-   C9 65       CMP   #$65
D04D-   F0 0E       BEQ   $D05D     ; REBOOT
D04F-   0A          ASL
D050-   10 14       BPL   $D066     ; $8x Commands (außer $82)
D052-   B0 1D       BCS   $D071     ; $Cx,$Dx Commands
D054-   4A          LSR
D055-   29 03       AND   #$03      ; $4x-Commands bleiben übrig
D057-   20 F3 D0    JSR   $D0F3     ; $4x COMMAND HANDLER
D05A-   4C 78 D0    JMP   $D078     ; Exit
D05D-   4C 03 BF    JMP   $BF03     ; REBOOT COMMAND HANDLER (kein Return)
D060-   20 06 BF    JSR   $BF06     ; $82 COMMAND HANDLER - Vorstufe
D063-   4C 78 D0    JMP   $D078     ; Exit
```

### – $80/$81 COMMAND HANDLER – Vorstufe

```
D066-   4A          LSR
D067-   69 01       ADC   #$01
D069-   85 42       STA   $42          ; $80=>$01  $81=>$02
D06B-   20 B2 D0    JSR   $D0B2        ; eigentlicher COMMAND HANDLER
D06E-   4C 78 D0    JMP   $D078        ; Exit
```

### – $Cx,Dx COMMAND HANDLER – Vorstufe

```
D071-   4A          LSR
D072-   29 1F       AND   #$1F         ; Bereich: $00-$13 für $C0-$D3
D074-   AA          TAX
D075-   20 3C D2    JSR   $D23C        ; eigentlicher COMMAND HANDLER

D085-   4E 9B BF    LSR   $BF9B        ; Reset MLIAKTIV
D088-   68          PLA                ; P-reg (Carry!)
D089-   AA          TAX
D08A-   AD 9D BF    LDA   $BF9D        ; Reload der Returnadresse
D08D-   48          PHA                  zum „user"-Programm
D08E-   AD 9C BF    LDA   $BF9C
D091-   48          PHA
D092-   8A          TXA                ; P-reg (Carry)
D093-   48          PHA
D094-   98          TYA
D095-   AE 9E BF    LDX   $BF9E        ; XSAVE
D098-   AC 9F BF    LDY   $BF9F        ; YSAVE
D09B-   48          PHA                ; ERR# (0 = kein Fehler)
D09C-   AD F4 BF    LDA   $BFF4        ; BANKSEL - welche Bank war vorher an?
D09F-   4C A0 BF    JMP   $BFA0        ; MLIEXIT
```

MLIEXIT schaltet auf die Konfiguration zurück (Bank/ROM), die vor dem MLI-Aufruf aktiv war. Die ERR# wird vom Stack geholt und ein RTI zum „user"-Programm ausgeführt. Bei dem RTI wird das P-Reg ebenfalls vom Stack geholt.

### – NO DEVICE

Alle „unbesetzten" Disk Driver-Startadressen in der Global Page zeigen hierher.

```
D0A2-   A9 28       LDA   #$28         ; ERR #28: NO DEVICE CONNECTED
D0A4-   20 09 BF    JSR   $BF09        ; SET SYSERR - RTS zur Routine, die $D0DA
                                         aufgerufen hat

D0A7-   A9 01       LDA   #$01         ; ERR #01: BAD SYSTEM CALL NUMBER
D0A9-   D0 02       BNE   $D0AD
D0AB-   A9 04       LDA   #$04         ; ERR #04: BAD SYS CALL PARAM COUNT
D0AD-   20 D7 D0    JSR   $D0D7        ; SET SYSERR - kehrt hierher zurück
D0B0-   B0 C6       BCS   $D078        ; „always" - SET SYSERR setzt Carry
```

## – $8x ($81: READ / $82: WRITE) COMMANDS

Xref: $D06B – $80/$81 Command Handler Vorstufe

```
D0B2-   A0 05       LDY   #$05
D0B4-   08          PHP                 ; Save des INT-Status
D0B5-   78          SEI
D0B6-   B1 40       LDA   ($40),Y       ; UNITNO, BLOCKNO, MEMADDR aus dem
D0B8-   99 42 00    STA   $0042,Y         „user"-Programm auf die Zpage
D0BB-   88          DEY
D0BC-   D0 F8       BNE   $D0B6
D0BE-   A6 45       LDX   $45           ; Ziel/Quelladresse hi
D0C0-   86 4F       STX   $4F
D0C2-   E8          INX
D0C3-   E8          INX                 ; 1 Block: 2 Speicherseiten
D0C4-   A5 44       LDA   $44
D0C6-   F0 01       BEQ   $D0C9         ; Wenn der Block nicht auf $xx00 anfängt,
D0C8-   E8          INX                   dann geht er in eine dritte Seite hinein
D0C9-   20 89 EE    JSR   $EE89         ; Check mit System Bit Map, s. „Spezielles"
D0CC-   B0 08       BCS   $D0D6         ; Wenn belegt, ERR#56: BAD BUF ADDRESS
D0CE-   20 DA D0    JSR   $D0DA         ; DO DISK I/O via Global Page
D0D1-   B0 03       BCS   $D0D6         ; bei Fehler: ERR# durch RWTB gesetzt
D0D3-   28          PLP
D0D4-   18          CLC
D0D5-   60          RTS                 ; keine Fehler => $D078
D0D6-   28          PLP                 ; ERROR return
```

Xref: $D0AD - s.o.

```
D0D7-   20 09 BF    JSR   $BF09         ; SET SYSERR. Für $80/81: RTS zu $D078
```

– **DO DISK I/O**: Startadresse des Drivers via Unitnummer aus der System Global Page

Xref: $D0CE - $81/$82 COMMAND Handler
$DDFD – General R/W block in/aus Dir ($F600)/VBM ($F400)
$E076 – General R/W block in/aus Filebuffer
$E2DA – READ Function, Stufe 3
$E66B – Test „Drive ready" und WRITE PROTECTED

```
D0DA-   A5 43       LDA   $43           ; UNITNO
D0DC-   29 F0       AND   #$F0          ; immer DSSS 0000
D0DE-   85 43       STA   $43
D0E0-   4A          LSR
D0E1-   4A          LSR                 ; /8 => 000D SSS0
D0E2-   4A          LSR                 ; => Index (* 2 für 16 Bit-Adresse)
D0E3-   AA          TAX
D0E4-   BD 10 BF    LDA   $BF10,X       ; in der DRIVER TABLE
D0E7-   8D B5 F0    STA   $F0B5
D0EA-   BD 11 BF    LDA   $BF11,X
D0ED-   8D B6 F0    STA   $F0B6
D0F0-   6C B5 F0    JMP   ($F0B5)       ; Sprung zur RWTB. RTS von da zum Caller
```

## – $4x ($40: ALLOCATE INT / $41: DEALLOCATE INT) COMMANDS

Xref: $D057 - Command dispatcher

```
D0F3-   85 42       STA   $42           ; Aufruf mit 1($40) oder 2($41) im Akku
D0F5-   4A          LSR
D0F6-   B0 2C       BCS   $D124         ; $41: DEALLOC INT
```

```
D0F8-   A2 03      LDX   #$03          ; ansonsten Start mit erstem Eintrag
D0FA-   BD 7E BF   LDA   $BF7E,X       ; in INT TABLE
D0FD-   D0 16      BNE   $D115         ; Sprung wenn besetzt
D0FF-   A0 03      LDY   #$03          ; Freier Eintrag gefunden:
D101-   B1 40      LDA   ($40),Y       ; Startadresse hi aus dem „user"-Programm
D103-   F0 1A      BEQ   $D11F         ; = 0  => ERR #53: INVALID PARAMETER
D105-   9D 7E BF   STA   $BF7E,X
D108-   88         DEY
D109-   B1 40      LDA   ($40),Y       ; Startadresse lo
D10B-   9D 7D BF   STA   $BF7D,X
D10E-   8A         TXA                 ; Index zur 16-Bit-Adresse: Nummer * 2
D10F-   4A         LSR                 ; / 2
D110-   88         DEY
D111-   91 40      STA   ($40),Y       ; zurückgelieferte Reference Number
D113-   18         CLC
D114-   60         RTS                 ; => $D078

D115-   E8         INX                 ; Eintrag ist besetzt. Nächster Index
D116-   E8         INX
D117-   E0 0B      CPX   #$0B          ; Table zu Ende?
D119-   D0 DF      BNE   $D0FA         ; noch nicht
D11B-   A9 25      LDA   #$25          ; ERR #25: INTERRUPT TABLE FULL
D11D-   D0 02      BNE   $D121
D11F-   A9 53      LDA   #$53          ; ERR #53: INVALID SYS CALL PARAM
D121-   20 09 BF   JSR   $BF09         ; SET SYSERR: RTS zu $D078
```

## – $41 COMMAND: DEALLOC INTERRUPT

```
D124-   A0 01      LDY   #$01
D126-   B1 40      LDA   ($40),Y       ; Reference Number aus dem „user"-Programm
D128-   F0 F5      BEQ   $D11F           muß im Bereich 1-4 sein
D12A-   C9 05      CMP   #$05
D12C-   B0 F1      BCS   $D11F         ; sonst ERR #53: INVALID SYS CALL PARM
D12E-   0A         ASL                 ; o.k: Reference Number * 2 als Index
D12F-   AA         TAX
D130-   A9 00      LDA   #$00
D132-   9D 7E BF   STA   $BF7E,X       ; Löschen des Eintrags ohne vorherige
D135-   9D 7F BF   STA   $BF7F,X         Prüfung, ob er überhaupt besetzt war
D138-   18         CLC
D139-   60         RTS                 ; => $D078
```

**IRQ HANDLER;** siehe Kapitel „Interrupts/RESET"

```
D13A-   A5 45      LDA   $45           ; „Monitor-IRQ": Akku hier gespeichert
D13C-   8D 88 BF   STA   $BF88         ; INTAREG
D13F-   8E 89 BF   STX   $BF89         ; INTXREG
D142-   8C 8A BF   STY   $BF8A         ; INTYREG
D145-   BA         TSX
D146-   8E 8B BF   STX   $BF8B         ; INTSREG
D149-   68         PLA
D14A-   8D 8C BF   STA   $BF8C         ; INTPREG - Statusregister
D14D-   68         PLA
D14E-   8D 8E BF   STA   $BF8E
D151-   68         PLA                 ; INTADDR - Returnadresse
D152-   8D 8F BF   STA   $BF8F
D155-   9A         TXS                 ; Stackpointer wieder auf Anfangsstand
D156-   AD F8 07   LDA   $07F8         ; Wenn eine Karte „an" ist, ist hier
D159-   8D C4 D1   STA   $D1C4           die Slotadresse (hi) gespeichert
D15C-   BA         TSX
D15D-   30 09      BMI   $D168         ; Sprung wenn Stackpointer > $7F
D15F-   A0 0F      LDY   #$0F
```

```
D161-  68          PLA             ; sonst werden 16 Werte vom Stack geholt
D162-  99 AF F0    STA  $F0AF,Y      und gespeichert („Platz machen")
D165-  88          DEY               Dabei werden $F0B5/B6 überschrieben,
D166-  10 F9       BPL  $D161        (s. „Spezielles")
D168-  A2 FA       LDX  #$FA
D16A-  B5 00       LDA  $00,X      ; „Retten" von $FA-$FF auf der Zpage
D16C-  9D A5 EF    STA  $EFA5,X
D16F-  E8          INX
D170-  D0 F8       BNE  $D16A

D172-  AD 81 BF    LDA  $BF81      ; Service 1: Hibyte der Startadresse
D175-  F0 05       BEQ  $D17C      ; unbesetzt: Sprung
D177-  20 CE D1    JSR  $D1CE      ; besetzt: Aufruf. SEC wenn nicht zuständig
D17A-  90 23       BCC  $D19F
D17C-  AD 83 BF    LDA  $BF83      ; Service 2: Hibyte der Startadresse
D17F-  F0 05       BEQ  $D186
D181-  20 D1 D1    JSR  $D1D1      ; besetzt: Aufruf. SEC wenn nicht zuständig
D184-  90 19       BCC  $D19F
D186-  AD 85 BF    LDA  $BF85      ; Service 3
D189-  F0 05       BEQ  $D190
D18B-  20 D4 D1    JSR  $D1D4      ; besetzt: Aufruf. SEC wenn nicht zuständig
D18E-  90 0F       BCC  $D19F
D190-  AD 87 BF    LDA  $BF87      ; Service 4
D193-  F0 05       BEQ  $D19A
D195-  20 D7 D1    JSR  $D1D7      ; besetzt: Aufruf. SEC wenn nicht zuständig
D198-  90 05       BCC  $D19F

D19A-  A9 01       LDA  #$01       ; keine der Service-Routinen zuständig:
D19C-  20 0C BF    JSR  $BF0C      ; SYSTEM DEATH mit ERR# 1

D19F-  A2 FA       LDX  #$FA       ; ok. Restore der Zpage
D1A1-  BD A5 EF    LDA  $EFA5,X
D1A4-  95 00       STA  $00,X
D1A6-  E8          INX
D1A7-  D0 F8       BNE  $D1A1
D1A9-  AE 8B BF    LDX  $BF8B      ; INTSREG: wurde Platz gemacht?
D1AC-  30 0B       BMI  $D1B9      ; Nein, war genug Platz
D1AE-  A0 00       LDY  #$00       ; ansonsten Zurückladen
D1B0-  B9 AF F0    LDA  $F0AF,Y
D1B3-  48          PHA
D1B4-  C8          INY
D1B5-  C0 10       CPY  #$10
D1B7-  D0 F7       BNE  $D1B0
D1B9-  AC 8A BF    LDY  $BF8A      ; INTYREG
D1BC-  AE 89 BF    LDX  $BF89      ; INTXREG
D1BF-  AD FF CF    LDA  $CFFF      ; CLRROM - Abschalten aller Karten
D1C2-  AD 00 C1    LDA  $C100      ; REACTIVATE CARD: $D1C4 von $D159 auf
D1C5-  AD C4 D1    LDA  $D1C4        Slotno (hi) gesetzt
D1C8-  8D F8 07    STA  $07F8      ; Restore $07F8
D1CB-  4C D0 BF    JMP  $BFD0      ; INTEXIT in Global Page
```

Xref: Nur von INT Service via JSR

```
D1CE-  6C 80 BF    JMP  ($BF80)    ; Start der INT-Services
D1D1-  6C 82 BF    JMP  ($BF82)
D1D4-  6C 84 BF    JMP  ($BF84)
D1D7-  6C 86 BF    JMP  ($BF86)
```

– **SET SYSERR:** Sprung hierher von $BF09

```
D1DA-   8D 0F BF    STA   $BF0F          ; ERR#
D1DD-   68          PLA
D1DE-   68          PLA                  ; eine Returnadresse „rauswerfen"
D1DF-   AD 0F BF    LDA   $BF0F
D1E2-   38          SEC                  ; Indikator: ERROR
D1E3-   60          RTS                  ; Exit (normalerweise zu $D078)
```

– **SYSTEM DEATH**: Entry Nummer 2: Setzt ERR# = 0 (s. „Global Page", $BF03/$BFF6)

```
D1E4-   A9 00       LDA   #$00
```

– **SYSTEM DEATH**: Entry Nummer 1: Aufruf mit ERR# in A. Sprung hierher von $BF0C

```
D1E6-   AA          TAX                  ; ERR#
D1E7-   8D 0C C0    STA   $C00C          ; Abschalten der 80-Zeichen-Karte (IIc/e)
D1EA-   AD 51 C0    LDA   $C051          ; TEXT
D1ED-   AD 53 C0    LDA   $C053          ; MIX
D1F0-   AD 54 C0    LDA   $C054          ; PAGE 1
D1F3-   AD 56 C0    LDA   $C056          ; LORES
D1F6-   A0 27       LDY   #$27
D1F8-   A9 A0       LDA   #$A0           ; Space
D1FA-   99 50 07    STA   $0750,Y
D1FD-   B9 DE EF    LDA   $EFDE,Y        ; INSERT SYSTEM DISK AND RESTART
D200-   99 D0 07    STA   $07D0,Y        ; direkt in den Bildspeicher
D203-   88          DEY
D204-   10 F2       BPL   $D1F8
D206-   8A          TXA                  ; ERR#
D207-   F0 30       BEQ   $D239          ; 00: „-ERRxx" wird nicht gedruckt
D209-   A9 AD       LDA   #$AD
D20B-   8D F1 07    STA   $07F1          ; „-"
D20E-   A9 C5       LDA   #$C5
D210-   8D F2 07    STA   $07F2          ; „E"
D213-   A9 D2       LDA   #$D2
D215-   8D F3 07    STA   $07F3          ; „R"
D218-   8D F4 07    STA   $07F4          ; „R"
D21B-   8A          TXA                  ; ERR#
D21C-   4A          LSR
D21D-   4A          LSR
D21E-   4A          LSR
D21F-   4A          LSR                  ; Übersetzung des höheren Nibble
D220-   09 B0       ORA   #$B0
D222-   C9 BA       CMP   #$BA
D224-   90 02       BCC   $D228
D226-   69 06       ADC   #$06           ; auf „A"-„F" für Werte > 9
D228-   8D F6 07    STA   $07F6          ; „Print"
D22B-   8A          TXA
D22C-   29 0F       AND   #$0F           ; niederwertiges Nibble der Hexzahl
D22E-   09 B0       ORA   #$B0
D230-   C9 BA       CMP   #$BA
D232-   90 02       BCC   $D236
D234-   69 06       ADC   #$06           ; „A"-„F" für Werte > 9
D236-   8D F7 07    STA   $07F7          ; „Print"

D239-   4C 39 D2    JMP   $D239          ; bis zum nächsten RESET
```

## $Cx, $Dx COMMANDS

Xref: $D075 - Command dispatcher

```
D23C-  8E 77 F0    STX   $F077        ; Aufruf mit Index im Bereich $00-$13
D23F-  BD 8D EF    LDA   $EF8D,X      ; Bit-Table: Bit 7-5 bestimmen, welche
D242-  0A          ASL                  Routinen vorher aufgerufen werden
D243-  8D 73 F0    STA   $F073
D246-  29 3F       AND   #$3F         ; Bit 0-4 sind der Index (* 2) für die
D248-  AA          TAX                  Startadresse des eigentlichen
D249-  BD 65 EF    LDA   $EF65,X      ; Kommandos in der nächsten Tabelle
D24C-  8D B5 F0    STA   $F0B5
D24F-  BD 66 EF    LDA   $EF66,X      ; $F0B5/B6: Startadresse des Commands
D252-  8D B6 F0    STA   $F0B6
D255-  A9 20       LDA   #$20
D257-  8D 74 F0    STA   $F074        ; internes BACKUPBIT - wird immer gesetzt
D25A-  90 05       BCC   $D261        ; Carry vom ersten „ASL" ($D242)
D25C-  20 7F D2    JSR   $D27F        ; „Copy & Verify Pathname"
D25F-  B0 17       BCS   $D278        ; Error: => SET SYSERR => $D078
D261-  0E 73 F0    ASL   $F073
D264-  90 05       BCC   $D26B
D266-  20 C5 D3    JSR   $D3C5        ; „Set File Data"
D269-  B0 0D       BCS   $D278        ; Error: => SET SYSERR => $D078
D26B-  0E 73 F0    ASL   $F073
D26E-  90 03       BCC   $D273
D270-  20 06 BF    JSR   $BF06        ; SET TIME - keine Fehler möglich

D273-  20 7C D2    JSR   $D27C        ; Ausführung des Command
D276-  90 03       BCC   $D27B        ; keine Fehler: RTS zu $D078

D278-  20 09 BF    JSR   $BF09        ; SET SYSERR. RTS zu $D078
D27B-  60          RTS
```

Xref: $D273 – $C0-$D3 Commands

```
D27C-  6C B5 F0    JMP   ($F0B5)      ; ersetzt „JSR (xx)"
```

**– COPY & VERIFY PATHNAME**

Aufruf vor CREATE/DESTROY/RENAME/SET FILE INFO/GET FILE INFO/OPEN
Der Pathname wird aus dem Parameterblock des „user"-Programms geholt, nach $F100... kopiert und verifiziert. Dabei wird auf Großbuchstaben übersetzt und auf illegale Zeichen geprüft. Das erste Zeichen jedes Namens muß ein Buchstabe sein, die Länge des gesamten Namens darf $40 nicht überschreiten.
Der Path erhält dabei die folgende Form:
Länge „NAME1" Länge „NAME2" ... Länge „NAMEn" 00 <Ende>

Xref: $D25C – $C0-$D3 Commands
$D32A – SET PREFIX (interner Aufruf – ERR#40 für PREFIX „/" wird abgefangen)
$EAB2 – RENAME (vor der Suche des „alten" Files)

```
D27F-  A0 01       LDY   #$01
D281-  B1 40       LDA   ($40),Y      ; Pointer zum Pathname aus dem „user"-
D283-  85 48       STA   $48            Programm auf $0048/49
D285-  C8          INY
D286-  B1 40       LDA   ($40),Y
D288-  85 49       STA   $49
```

Xref: JP $EB40 – RENAME ($0048/49 auf den „neuen" Path gesetzt)

```
D28A-   A2 00       LDX   #$00
D28C-   A0 00       LDY   #$00
D28E-   8E 7C F0    STX   $F07C         ; Reset FULLNAME FLAG
D291-   8E 00 F1    STX   $F100         ; Länge des Pathname jetzt: Null
D294-   B1 48       LDA   ($48),Y       ; Länge des gesamten Pathname
D296-   F0 58       BEQ   $D2F0         ; 0: ERR #40: INVALID PATHNAME
D298-   C9 41       CMP   #$41
D29A-   B0 54       BCS   $D2F0         ; >$40: ERR #40: INVALID PATHNAME
D29C-   8D 5E F0    STA   $F05E
D29F-   EE 5E F0    INC   $F05E         ; max. Index
D2A2-   C8          INY
D2A3-   B1 48       LDA   ($48),Y
D2A5-   09 80       ORA   #$80
D2A7-   C9 AF       CMP   #$AF          ; Start mit „/"?
D2A9-   D0 04       BNE   $D2AF
D2AB-   8D 7C F0    STA   $F07C         ; wenn ja, set FULLNAME FLAG
D2AE-   C8          INY                 ; das „/" wird in diesem Fall übersprungen

D2AF-   A9 FF       LDA   #$FF          ; Indikator: erstes Zeichen im Namen
D2B1-   9D 00 F1    STA   $F100,X
D2B4-   8D 78 F0    STA   $F078         ; und Längenzähler für einen Namen im Path
D2B7-   8E 7A F0    STX   $F07A         ; Start-Index für diesen Namen im Path
D2BA-   CC 5E F0    CPY   $F05E         ; Ende des gesamten Path?
D2BD-   B0 35       BCS   $D2F4
D2BF-   B1 48       LDA   ($48),Y       ; ein Zeichen
D2C1-   29 7F       AND   #$7F          ; Bit 7 wird maskiert
D2C3-   E8          INX
D2C4-   C8          INY
D2C5-   C9 2F       CMP   #$2F          ; „/"?
D2C7-   F0 40       BEQ   $D309         ; wenn ja, Ende des Namens (nicht des Path!)
D2C9-   C9 61       CMP   #$61
D2CB-   90 02       BCC   $D2CF
D2CD-   29 5F       AND   #$5F          ; wenn >„a", Übersetzung auf Großschrift
D2CF-   9D 00 F1    STA   $F100,X
D2D2-   EE 78 F0    INC   $F078         ; Start mit $FF als Flag für „erstes Zeichen"
D2D5-   D0 05       BNE   $D2DC
D2D7-   EE 78 F0    INC   $F078         ; wenn dies das erste Zeichen im Namen war,
D2DA-   D0 0C       BNE   $D2E8           Index + 2 und nur Test auf Buchstabe
D2DC-   C9 2E       CMP   #$2E
D2DE-   F0 DA       BEQ   $D2BA         ; „." ist ok: NE
D2E0-   C9 30       CMP   #$30
D2E2-   90 0C       BCC   $D2F0         ; < „0": INVALID PATHNAME
D2E4-   C9 3A       CMP   #$3A
D2E6-   90 D2       BCC   $D2BA         ; „0"-„9" ist ok: NE
D2E8-   C9 41       CMP   #$41
D2EA-   90 04       BCC   $D2F0         ; > „9" and < „A": INVALID PATHNAME
D2EC-   C9 5B       CMP   #$5B
D2EE-   90 CA       BCC   $D2BA         ; „A"-„Z" ist ok: NE
D2F0-   38          SEC                 ; ERR #40: INVALID PATHNAME
D2F1-   A9 40       LDA   #$40
D2F3-   60          RTS

D2F4-   A9 00       LDA   #$00          ; Ende eines Namens im Path
D2F6-   2C 78 F0    BIT   $F078         ; Länge dieses Namens: $FF, wenn der
D2F9-   10 04       BPL   $D2FF           Name nur aus „/" besteht => Längenbyte
D2FB-   8D 78 F0    STA   $F078         auf 00 (kein Fehler!)
D2FE-   CA          DEX             *   ; auf $FF wenn der GESAMTE Name nur
D2FF-   E8          INX             *     aus „/" besteht
D300-   9D 00 F1    STA   $F100,X       ; 00 nach dem Namen
D303-   F0 EB       BEQ   $D2F0     *   ; Z-flag von INX! : ERR #40
D305-   8E 5E F0    STX   $F05E         ; Gesamtlänge des Pathname
D308-   AA      *   TAX                 ; = 0 !
```

```
D309-  AD 78 F0     LDA  $F078        ; Einsprung, wenn „/" gefunden: Länge
D30C-  C9 10        CMP  #$10
D30E-  B0 E0        BCS  $D2F0        ; dieser Name ist länger als $0F Zeichen
D310-  8E 7D F0 *   STX  $F07D        ; Save des momentanen Index
D313-  AE 7A F0     LDX  $F07A        ; Reload Startindex dieses Namens im Path
D316-  9D 00 F1     STA  $F100,X      ; Speichern der Länge als erstes Byte
D319-  AE 7D F0 *   LDX  $F07D        ; Reload momentanen Index
D31C-  D0 91    *   BNE  $D2AF        ; wenn von $D2F4: Ende
D31E-  18           CLC
D31F-  AD 7C F0     LDA  $F07C        ; FULLNAME FLAG: vollständiger Name?
D322-  D0 05        BNE  $D329        ; Ja. Kein System Prefix dazu
D324-  AD 9A BF     LDA  $BF9A        ; ansonsten check PREFIX
D327-  F0 C7        BEQ  $D2F0        ; kein PREFIX => ERR #40
D329-  60           RTS
```

## $C6 – SET PREFIX COMMAND; keine Routinen vorher aufgerufen

```
D32A-  20 7F D2     JSR  $D27F        ; Copy & Verify Pathname
D32D-  90 0A        BCC  $D339
D32F-  AC 00 F1     LDY  $F100        ; Länge des ersten Namens im Path
D332-  D0 4C        BNE  $D380        ; <> 0 und ERR => EXIT mit ERR#
D334-  8C 9A BF     STY  $BF9A        ; ist 0: Reset System PREFIX
D337-  18           CLC               ; das ist ein gültiges Command!
D338-  60           RTS

D339-  20 98 D7     JSR  $D798        ; normaler Pathname: GET FILE
D33C-  90 04        BCC  $D342
D33E-  C9 40        CMP  #$40         ; INVALID (Ende in VolDIR) ist kein Fehler
D340-  D0 3E        BNE  $D380        ; ansonsten Exit mit ERR#
D342-  AD 1F F0     LDA  $F01F        ; Storage Type des Files –
D345-  29 D0        AND  #$D0         ; muß eine (Sub)directory sein
D347-  49 D0        EOR  #$D0
D349-  D0 33        BNE  $D37E        ; ERR #4B: UNSUPPORTED STORAGE TYPE
D34B-  AC 7C F0     LDY  $F07C        ; FULLNAME FLAG
D34E-  D0 03        BNE  $D353
D350-  AD 9A BF     LDA  $BF9A        ; sonst Start mit Index für SYS PREFIX
D353-  A8           TAY               ; 0 für „Full name", sonst Index von PREFIX
D354-  38           SEC
D355-  ED 5E F0     SBC  $F05E        ; minus Länge des neuen Namens
D358-  C9 C0        CMP  #$C0         ; alles zusammen länger als $40 Zeichen?
D35A-  90 94        BCC  $D2F0        ; Exit mit ERR #40
D35C-  AA           TAX
D35D-  8D 9A BF     STA  $BF9A        ; sonst Update von PREFIX zum neuen Start
D360-  AD 19 F0     LDA  $F019        ; Unitnummer des PREFIX-Units
D363-  8D 5F F0     STA  $F05F        ; (nie mehr benutzt)
D366-  AD 30 F0     LDA  $F030        ; Nummer des Key Blocks der PREFIX-Dir
D369-  8D 60 F0     STA  $F060
D36C-  AD 31 F0     LDA  $F031        ; PREFIX DIRECTORY
D36F-  8D 61 F0     STA  $F061
D372-  B9 00 F1     LDA  $F100,Y      ; 0, wenn FULLNAME, ansonsten Start mit
                                      ; altem PREFIX und OVERFLOW in den neuen
D375-  9D 00 F1     STA  $F100,X      ; Namen, d.h. Anhängen des neuen Namens
D378-  C8           INY
D379-  E8           INX               ; bis X (nicht Y!) = 0
D37A-  D0 F6        BNE  $D372        ; der neue (Gesamt)name fängt im Bereich
D37C-  18           CLC               ; $F1C0..$F1FE an
D37D-  60           RTS               ; => $D078

D37E-  A9 4B        LDA  #$4B         ; ERR #4B: UNSUPPORTED STORAGE TYPE
D380-  38           SEC
D381-  60           RTS
```

**$C7 – GET PREFIX COMMAND;** keine anderen Routinen vorher

```
D382-   18          CLC
D383-   A0 01       LDY   #$01          ; Startadresse des „user"-Puffers
D385-   B1 40       LDA   ($40),Y         wird auf $004E/4F kopiert
D387-   85 4E       STA   $4E
D389-   C8          INY
D38A-   B1 40       LDA   ($40),Y
D38C-   85 4F       STA   $4F
D38E-   A9 00       LDA   #$00
D390-   8D 9B F0    STA   $F09B
D393-   A9 40       LDA   #$40          ; Platzbedarf: $40 Zeichen (max.)
D395-   8D 9A F0    STA   $F09A
D398-   20 6C EE    JSR   $EE6C         ; Check mit System Bit Map
D39B-   B0 E3       BCS   $D380         ; ERR #56: BAD BUFFER ADDRESS
D39D-   A0 00       LDY   #$00
D39F-   AD 9A BF    LDA   $BF9A         ; PREFIX
D3A2-   AA          TAX
D3A3-   F0 04       BEQ   $D3A9         ; 0: kein Prefix (gültig!)
D3A5-   49 FF       EOR   #$FF
D3A7-   69 02       ADC   #$02          ; Längenberechung vom Startindex ($F1C0-FF)
D3A9-   91 4E       STA   ($4E),Y       ; erstes Byte im „user"-Puffer
D3AB-   F0 16       BEQ   $D3C3         ; wenn 0 => Kopie beendet
D3AD-   C8          INY
D3AE-   BD 00 F1    LDA   $F100,X
D3B1-   91 4E       STA   ($4E),Y       ; Kopie in den „user"-Puffer
D3B3-   29 F0       AND   #$F0
D3B5-   D0 04       BNE   $D3BB         ; wenn dabei 0 herauskommt, ist das ein
D3B7-   A9 2F       LDA   #$2F            Längenbyte ($00-$0F) und wird durch „/"
D3B9-   D0 F6       BNE   $D3B1           ersetzt
D3BB-   E8          INX
D3BC-   D0 EF       BNE   $D3AD         ; Loop bis Index = 00
D3BE-   C8          INY
D3BF-   A9 2F       LDA   #$2F          ; ein letzter „/" wird angehängt
D3C1-   91 4E       STA   ($4E),Y
D3C3-   18          CLC
D3C4-   60          RTS                 ; => $D078
```

**– SET FILE DATA**

Aufruf vor NEWLINE/READ/WRITE/SET & GET: MARK/EOF/BUF
Überprüfung der Reference Number und ob der Eintrag im FCB existiert.
Setzt Filebuffer-Adresse, VCB-Index und berechnet den FCB-Index.
Existiert kein VCB-Eintrag/kein Filebuffer für ein OPEN File, folgt SYSTEM DEATH.

Xref: $D266 – $C0-$D3 Commands
$E721 – FLUSH (single file – das Argument $00 (ALL) wird vorher abgefangen)

```
D3C5-   A0 01       LDY   #$01
D3C7-   B1 40       LDA   ($40),Y       ; Reference Number vom „user"
D3C9-   F0 5B       BEQ   $D426
D3CB-   C9 09       CMP   #$09          ; muß im Bereich 1-8 sein
D3CD-   B0 57       BCS   $D426         ; ERR#43: INVALID REFERENCE NUMBER
D3CF-   48          PHA
D3D0-   E9 00       SBC   #$00          ; minus 1 => 0-7
D3D2-   4A          LSR
D3D3-   6A          ROR
D3D4-   6A          ROR
```

```
D3D5-  6A          ROR
D3D6-  8D 52 F0    STA   $F052          ; *$20 => FCB-Index
D3D9-  A8          TAY
D3DA-  68          PLA
D3DB-  D9 00 F3    CMP   $F300,Y        ; der Eintrag muß die Reference Number
D3DE-  D0 41       BNE   $D421            enthalten, d.h. existieren
```

– Laden der FCB-Parameter und Setzen der VCB-Index

Xref: $E14F – OPEN (Setzen der Parameter für das Lesen des ersten Blocks – SET MARK)
JP $E3AE – READ (Re-set der Filebuffer-Pointer nach Stufe 2/3)
$E714 $- FLUSH (ein File im FLUSH ALL-Loop)

```
D3E0-  B9 0B F3    LDA   $F30B,Y        ; Puffernummer * 2
D3E3-  20 26 EE    JSR   $EE26          ; Pufferstart von SGP => F09D/9E
D3E6-  AE 9E F0    LDX   $F09E          ; kein Puffer (hi-Adresse = 0)?
D3E9-  F0 27       BEQ   $D412          ; SYSTEM DEATH #0B
D3EB-  86 4B       STX   $4B            ; Start des Datenteils
D3ED-  E8          INX
D3EE-  E8          INX
D3EF-  86 49       STX   $49            ; Indexteil ($200 darüber)
D3F1-  B9 01 F3    LDA   $F301,Y        ; Unitnummer
D3F4-  8D 30 BF    STA   $BF30
D3F7-  AD 9D F0    LDA   $F09D
D3FA-  85 4A       STA   $4A            ; Pufferstarts lo (immer 00)
D3FC-  85 48       STA   $48
D3FE-  AA          TAX
D3FF-  BD 10 F2    LDA   $F210,X        ; Suche nach dem VCB-Eintrag, d.h. dem
D402-  D9 01 F3    CMP   $F301,Y          Unit, auf dem der File existiert
D405-  F0 10       BEQ   $D417          ; gefunden
D407-  8A          TXA
D408-  18          CLC
D409-  69 20       ADC   #$20           ; nächster VCB-Eintrag
D40B-  90 F1       BCC   $D3FE
D40D-  A9 0A       LDA   #$0A           ; KEIN UNIT FÜR DEN FILE!
D40F-  20 0C BF    JSR   $BF0C            SYSTEM DEATH #0A
D412-  A9 0B       LDA   #$0B           ; KEIN BUFFER FÜR DEN FILE!
D414-  20 0C BF    JSR   $BF0C            SYSTEM DEATH #0B

D417-  BD 00 F2    LDA   $F200,X        ; VCB-Eintrag mit passender Unitno gefunden
D41A-  F0 EB       BEQ   $D407          ; wenn ungültig, nächster Eintrag
D41C-  8E 51 F0    STX   $F051          ; VCB-Index
D41F-  18          CLC
D420-  60          RTS                  ; zum Aufrufer

D421-  A9 00       LDA   #$00           ; ERR Exit: kein FCB-Eintrag
D423-  99 52 F0    STA   $F052,Y ****   ; BUG: s. „Spezielles"
D426-  A9 43       LDA   #$43           ; ERR#43: INVALID REFERENCE NUMBER
D428-  38          SEC
D429-  60          RTS                  ; zum Aufrufer
```

**$C5 – ONLINE COMMAND;** keine anderen Routinen vorher

```
D42A-  20 03 E4    JSR   $E403          ; Startadr. des „user"-Puffers => $004E/4F
D42D-  A9 00       LDA   #$00
D42F-  8D 9A F0    STA   $F09A
D432-  8D 9B F0    STA   $F09B          ; Annahme: benötigter Platz = 00 00
D435-  A0 01       LDY   #$01
```

```
D437-   B1 40       LDA   ($40),Y       ; Reference Number
D439-   F0 07       BEQ   $D442         ; 0 = ALL
D43B-   A9 10       LDA   #$10
D43D-   8D 9A F0    STA   $F09A         ; Spezifische Refno: $10 Bytes Platz
D440-   D0 05       BNE   $D447
D442-   A9 01       LDA   #$01          ; „ALL": $100 Bytes Platz werden gebraucht
D444-   8D 9B F0    STA   $F09B
D447-   20 6C EE    JSR   $EE6C         ; Check mit System Bit Map
D44A-   B0 33       BCS   $D47F         ; ERR #56: BAD BUFFER ADDRESS
D44C-   A9 00       LDA   #$00
D44E-   AC 9A F0    LDY   $F09A
D451-   88          DEY
D452-   91 4E       STA   ($4E),Y       ; der Puffer ($10 oder $100 Bytes)
D454-   D0 FB       BNE   $D451           wird gelöscht
D456-   8D 7A F0    STA   $F07A         ; erster Index im Puffer: 00
D459-   A0 01       LDY   #$01
D45B-   B1 40       LDA   ($40),Y       ; Reference Number
D45D-   29 F0       AND   #$F0
D45F-   D0 1F       BNE   $D480         ; Spezifisch: Sprung zu ONLINE SPECIFIC
D461-   20 57 DA    JSR   $DA57         ; ALL: Alle DEVICES aus der SGP => $F08A...

D464-   8E 7D F0    STX   $F07D         ; DEVNUM: Anzahl der DEVICES
D467-   BD 8A F0    LDA   $F08A,X
D46A-   20 80 D4    JSR   $D480         ; Für jedes DEVICE ein ONLINE SPECIFIC
D46D-   AD 7A F0    LDA   $F07A
D470-   18          CLC
D471-   69 10       ADC   #$10          ; nächster Index im Puffer
D473-   8D 7A F0    STA   $F07A
D476-   AE 7D F0    LDX   $F07D         ; nächstes DEVICE
D479-   CA          DEX
D47A-   10 E8       BPL   $D464
D47C-   A9 00       LDA   #$00
D47E-   18          CLC
D47F-   60          RTS                 ; => $D078
```

– **ONLINE SPECIFIC;** ONLINE für ein Unit

Xref: $D45A – ONLINE COMMAND mit spezifizierter Reference Number
$D46A – ONLINE ALL Loop

```
D480-   8D 30 BF    STA   $BF30         ; Aufruf mit Unitnummer im Akku
D483-   20 69 DA    JSR   $DA69         ; Sucht nach diesem Unit im VCB oder nach
D486-   B0 30       BCS   $D4B8           einem freien Eintrag. ERR wenn VCB voll
D488-   A9 00       LDA   #$00            VCB-Index ($F051) ist auf gefundenen oder
D48A-   85 47       STA   $47             freien Eintrag gesetzt
D48C-   A9 02       LDA   #$02
D48E-   85 46       STA   $46
D490-   20 E1 DD    JSR   $DDE1         ; Der Key Block der VolDIR wird gelesen
```

```
D493-  B0 23      BCS  $D4B8      ; ERR - kein Exit!
D495-  AE 51 F0   LDX  $F051
D498-  BD 00 F2   LDA  $F200,X
D49B-  F0 05      BEQ  $D4A2      ; der VCB-Eintrag war frei, Sprung
D49D-  BD 11 F2   LDA  $F211,X    ; nicht frei: OPEN File?
D4A0-  30 0C      BMI  $D4AE      ; wenn ja, Vergleich neuer/alter Vol-Name

D4A2-  20 C4 DA   JSR  $DAC4      ; VCB-Eintrag wird mit neuer VolDIR besetzt
D4A5-  B0 11      BCS  $D4B8      ; ERR: kein ProDOS-Volume. Kein Exit!
D4A7-  A9 57      LDA  #$57       ; ERR#57: DUPLICATE VOLUME (kein Exit),
D4A9-  2C 75 F0   BIT  $F075        wenn $DA69 ein Volume mit OPEN File und
D4AC-  30 0A      BMI  $D4B8        gleichem Namen gefunden hat ($F075 = $80)

D4AE-  AE 51 F0   LDX  $F051      ; Vergleich alter VCB-Eintrag/neue VolDIR
D4B1-  20 1C DB   JSR  $DB1C        Nur sinnvoll, wenn Sprung von $D4A0
D4B4-  A9 2E      LDA  #$2E       ; ERR#2E: DISK SWITCHED AND FILE OPEN
D4B6-  90 1E      BCC  $D4D6      ; Sprung, wenn ok. ERR# ist dann ungültig
```

## – ERR HANDLER FÜR ONLINE

```
D4B8-  48         PHA             ; ERR#
D4B9-  20 EB D4   JSR  $D4EB      ; Unitnummer in den „user"-Puffer
D4BC-  68         PLA
D4BD-  C8         INY
D4BE-  91 4E      STA  ($4E),Y    ; und ERR# danach
D4C0-  C9 57      CMP  #$57       ; war ERR#: DUPLICATE VOLUME?
D4C2-  D0 10      BNE  $D4D4
D4C4-  C8         INY
D4C5-  AE 76 F0   LDX  $F076      ; Unitnr. des doppelten Vols, von $DA69
D4C8-  BD 10 F2   LDA  $F210,X
D4CB-  91 4E      STA  ($4E),Y    ; als zweites Byte in den „user"-Puffer
D4CD-  A9 00      LDA  #$00
D4CF-  8D 75 F0   STA  $F075      ; Reset des OPEN File-Flags
D4D2-  A9 57      LDA  #$57
D4D4-  38         SEC
D4D5-  60         RTS             ; => $D078 oder ONLINE ALL-Loop
```

– keine Fehler in ONLINE SPECIFIC

```
D4D6-  BD 00 F2   LDA  $F200,X    ; Länge des Volume-Namens
D4D9-  8D 78 F0   STA  $F078      ; Zähler
D4DC-  AC 7A F0   LDY  $F07A      ; Start-Index im „user"-Puffer
D4DF-  BD 00 F2   LDA  $F200,X
D4E2-  91 4E      STA  ($4E),Y    ; der Name wird kopiert
D4E4-  E8         INX
D4E5-  C8         INY
D4E6-  CE 78 F0   DEC  $F078
D4E9-  10 F4      BPL  $D4DF
```

– Fügt die Unitnummer zur Namenslänge hinzu

Xref: $D4B9 – ONLINE err handler

```
D4EB-  AC 7A F0   LDY  $F07A      ; Start-Index im „user"-Puffer
D4EE-  AD 30 BF   LDA  $BF30
D4F1-  11 4E      ORA  ($4E),Y    ; das Längenbyte des Namens ($00-$0F)
D4F3-  91 4E      STA  ($4E),Y      bekommt die Unitnummer (DSSS 0000) dazu
D4F5-  18         CLC
D4F6-  60         RTS             ; => ONLINE ERR/$D078/ONLINE ALL-Loop
```

**$C0 – CREATE COMMAND;** vorher Copy & Verify Pathname ($D27F)

Xref: JP $D5A6 – CREATE selber: Restart nach Anhängen eines Subdirectory-Blocks

```
D4F7-   20 AB D7    JSR   $D7AB         ; FIND FILE ENTRY
D4FA-   B0 04       BCS   $D500         ; muß FILE NOT FOUND ergeben!
D4FC-   A9 47       LDA   #$47          ; ERR#47: DUPLICATE FILENAME
D4FE-   38          SEC
D4FF-   60          RTS                 ; => $D078
D500-   C9 46       CMP   #$46          ; FILE NOT FOUND?
D502-   D0 FA       BNE   $D4FE         ; andere Fehler werden nicht maskiert
D504-   A0 07       LDY   #$07
D506-   B1 40       LDA   ($40),Y       ; Storage Type vom „user"
D508-   C9 04       CMP   #$04
D50A-   90 04       BCC   $D510         ; „normales" File
D50C-   C9 0D       CMP   #$0D          ; Subdirectory?
D50E-   D0 10       BNE   $D520         ; ERR#4E: UNSUPPORTED STORAGE TYPE
D510-   AD 30 BF    LDA   $BF30         ; Unitnummer
D513-   20 5B E6    JSR   $E65B         ; Test auf WRITE PROTECTED
D516-   B0 0B       BCS   $D523
D518-   AD 5B F0    LDA   $F05B         ; von FIND FILE ENTRY: Nummer eines freien
D51B-   F0 07       BEQ   $D524           Dir-Eintrags. 0 = kein freier Eintrag
D51D-   4C B6 D5    JMP   $D5B6         ; freier Eintrag vorhanden

D520-   A9 4B       LDA   #$4B          ; ERR#4B: UNSUPPORTED STORAGE TYPE
D522-   38          SEC
D523-   60          RTS                 ; => $D078
```

**– APPEND A BLOCK TO SUBDIRECTORY**

– kein freier Eintrag von FIND FILE ENTRY gefunden. Danach Restart von CREATE

```
D524-   AD 06 F0    LDA   $F006         ; Parent Pointer der momentanen DIR
D527-   0D 07 F0    ORA   $F007           Eine Volume Directory hat den Pointer
D52A-   D0 04       BNE   $D530           auf 00 00 gesetzt
D52C-   A9 49       LDA   #$49          ; ERR#49: VOLUME DIRECTORY FULL
D52E-   38          SEC
D52F-   60          RTS                 ; => $D078
D530-   A5 46       LDA   $46           ; Nummer des momentanen (letzten) DirBlocks
D532-   48          PHA
D533-   A5 47       LDA   $47
D535-   48          PHA
D536-   20 A9 DC    JSR   $DCA9         ; ALLOCATE NEXT BLOCK
D539-   AA          TAX                 ; eventuelle ERR#, ansonsten Blocknummer
D53A-   68          PLA                   des neuen Blocks (low)
D53B-   85 47       STA   $47
D53D-   68          PLA
D53E-   85 46       STA   $46
D540-   8A          TXA
D541-   B0 E0       BCS   $D523         ; ERR# während ALLOCATE NEXT BLOCK
D543-   8D 02 F6    STA   $F602         ; die Nummer des neuen Blocks wird im
D546-   8C 03 F6    STY   $F603           „alten" Block als Forward Ref gesetzt
D549-   20 DD DD    JSR   $DDDD           der DirBlock wird zurückgeschrieben
D54C-   B0 D5       BCS   $D523
D54E-   A2 01       LDX   #$01
D550-   B5 46       LDA   $46,X         ; die Nummer des „alten" DirBlocks wird im
D552-   9D 00 F6    STA   $F600,X         „neuen" Block als Backward Ref gesetzt
D555-   BD 02 F6    LDA   $F602,X
```

```
D558-  95 46      STA   $46,X        ; und die Nummer des „neuen" Blocks
D55A-  CA         DEX                   (Forward Ref im „alten" Block) als
D55B-  10 F3      BPL   $D550           Parameter für die RWTB
D55D-  E8         INX
D55E-  8A         TXA
D55F-  9D 02 F6   STA   $F602,X       ; der gesamte „neue" Block wird bis auf
D562-  9D 00 F7   STA   $F700,X         die Backward Reference gelöscht
D565-  E8         INX
D566-  D0 F7      BNE   $D55F
D568-  20 DD DD   JSR   $DDDD        ; und auf das Volume geschrieben
D56B-  B0 B6      BCS   $D523
D56D-  AD 06 F0   LDA   $F006        ; Nummer des Parent Blocks
D570-  85 46      STA   $46
D572-  AD 07 F0   LDA   $F007
D575-  85 47      STA   $47          ; der Parent Block der Subdirectory
D577-  20 E1 DD   JSR   $DDE1        ; wird gelesen
D57A-  AE 08 F0   LDX   $F008        ; Eintragsnummer in diesem Block
D57D-  A9 F6      LDA   #$F6
D57F-  85 49      STA   $49
D581-  A9 04      LDA   #$04
D583-  18         CLC                ; $0048/49: Startadresse des Eintrags
D584-  CA         DEX
D585-  F0 09      BEQ   $D590
D587-  6D 09 F0   ADC   $F009
D58A-  90 F7      BCC   $D583
D58C-  E6 49      INC   $49
D58E-  B0 F3      BCS   $D583

D590-  85 48      STA   $48
D592-  A0 13      LDY   #$13         ; BLOCKS USED und EOF werden auf den
D594-  B1 48      LDA   ($48)          neuen Stand gebracht:
D596-  79 8E EF   ADC   $EF8E,Y        BLOCKS USED plus 1
D599-  91 48      STA   ($48),Y        EOF plus $200
D59B-  C8         INY
D59C-  98         TYA
D59D-  49 18      EOR   #$18         ; anstelle von CMP (Carry!)
D59F-  D0 F3      BNE   $D594
D5A1-  20 DD DD   JSR   $DDDD        ; der Parent Block wird zurückgeschrieben
D5A4-  B0 0F      BCS   $D5B5
D5A6-  4C F7 D4   JMP   $D4F7        ; Kompletter Restart von CREATE
```

– Löscht den gesamten Directory Space

Xref: $D5B6 – CREATE Subdir Key Block oder (leeren) Datenblock des neuen Files

```
D5A9-  A9 00      LDA   #$00
D5AB-  AA         TAX
D5AC-  9D 00 F6   STA   $F600,X
D5AF-  9D 00 F7   STA   $F700,X
D5B2-  E8         INX
D5B3-  D0 F7      BNE   $D5AC
D5B5-  60         RTS
```

– Erstellung eines Subdirectory Key oder eines leeren Datenblocks
Erstellung eines File Info (für den File-Eintrag in der Parent Directory)

Xref: $D51D – JP von CREATE, wenn ein freier Directory-Eintrag gefunden wurde

```
D5B6-   20 A9 D5    JSR   $D5A9          ; Löscht den Directory Space
D5B9-   A0 0B       LDY   #$0B
D5BB-   B1 40       LDA   ($40),Y        ; CREATE DATE  &  TIME vom „user"
D5BD-   99 2F F0    STA   $F02F,Y
D5C0-   8A          TXA
D5C1-   11 40       ORA   ($40),Y
D5C3-   AA          TAX                  ; ORA für alle vier Bytes
D5C4-   88          DEY
D5C5-   C0 07       CPY   #$07
D5C7-   D0 F2       BNE   $D5BB
D5C9-   8A          TXA                  ; Resultat des ORA
D5CA-   D0 0B       BNE   $D5D7          ; DATE & TIME war vom „user" spezifiziert
D5CC-   A2 03       LDX   #$03
D5CE-   BD 90 BF    LDA   $BF90,X        ; ansonsten wird die System Time kopiert
D5D1-   9D 37 F0    STA   $F037,X
D5D4-   CA          DEX
D5D5-   10 F7       BPL   $D5CE
D5D7-   B1 40       LDA   ($40),Y        ; Storage Type vom „user"
D5D9-   C9 04       CMP   #$04
D5DB-   A9 10       LDA   #$10           ; 0(!)/1/2/3 => $10
D5DD-   90 02       BCC   $D5E1
D5DF-   A9 D0       LDA   #$D0           ; $D0: Subdirectory
D5E1-   AE 7A F0    LDX   $F07A          ; Start-Index des letzten Namens im Path
D5E4-   1D 00 F1    ORA   $F100,X          (von FIND FILE ENTRY übriggeblieben)
D5E7-   8D 1F F0    STA   $F01F          ; = Storage Type/Namenslänge
D5EA-   29 0F       AND   #$0F
D5EC-   A8          TAY                  ; Namenslänge
D5ED-   18          CLC
D5EE-   6D 7A F0    ADC   $F07A
D5F1-   AA          TAX
D5F2-   BD 00 F1    LDA   $F100,X
D5F5-   99 1F F0    STA   $F01F,Y        ; der Name wird auf $F020-$F02E kopiert
D5F8-   CA          DEX
D5F9-   88          DEY
D5FA-   D0 F6       BNE   $D5F2
D5FC-   A0 03       LDY   #$03
D5FE-   B1 40       LDA   ($40),Y
D600-   8D 3D F0    STA   $F03D          ; $F03D: ACCESS
D603-   C8          INY
D604-   B1 40       LDA   ($40),Y
D606-   8D 2F F0    STA   $F02F          ; $F02F: FILE TYPE
D609-   C8          INY
D60A-   B1 40       LDA   ($40),Y
D60C-   99 39 F0    STA   $F039,Y        ; $F03E/3F: AUX TYPE
D60F-   C0 06       CPY   #$06
D611-   D0 F6       BNE   $D609
D613-   AD B0 EF    LDA   $EFB0
D616-   8D 3B F0    STA   $F03B          ; $F03B/3C: ProDOS VERSION (00 00)
D619-   AD B1 EF    LDA   $EFB1
D61C-   8D 3C F0    STA   $F03C
D61F-   A9 01       LDA   #$01
D621-   8D 32 F0    STA   $F032          ; $F032/33: BLOCKS USED (00 01)
D624-   AD 1A F0    LDA   $F01A
D627-   8D 44 F0    STA   $F044          ; $F044/45: Nummer des Key Blocks der
D62A-   AD 1B F0    LDA   $F01B            Directory, in dem dieser File steht
D62D-   8D 45 F0    STA   $F045
```

– File Info ist erstellt: Subdirectory Key oder leerer Datenblock?

```
D630-   AD 1F F0    LDA   $F01F          ; Storage Type/Namenslänge
D633-   29 E0       AND   #$E0
D635-   F0 37       BEQ   $D66E          ; Sprung für Type $10 (Datenfile)
D637-   A2 1E       LDX   #$1E
D639-   BD 1F F0    LDA   $F01F,X
```

```
D558-  95 46     STA  $46,X      ; und die Nummer des „neuen" Blocks
D55A-  CA        DEX              (Forward Ref im „alten" Block) als
D55B-  10 F3     BPL  $D550       Parameter für die RWTB
D55D-  E8        INX
D55E-  8A        TXA
D55F-  9D 02 F6  STA  $F602,X    ; der gesamte „neue" Block wird bis auf
D562-  9D 00 F7  STA  $F700,X      die Backward Reference gelöscht
D565-  E8        INX
D566-  D0 F7     BNE  $D55F
D568-  20 DD DD  JSR  $DDDD      ; und auf das Volume geschrieben
D56B-  B0 B6     BCS  $D523
D56D-  AD 06 F0  LDA  $F006      ; Nummer des Parent Blocks
D570-  85 46     STA  $46
D572-  AD 07 F0  LDA  $F007
D575-  85 47     STA  $47        ; der Parent Block der Subdirectory
D577-  20 E1 DD  JSR  $DDE1      ; wird gelesen
D57A-  AE 08 F0  LDX  $F008      ; Eintragsnummer in diesem Block
D57D-  A9 F6     LDA  #$F6
D57F-  85 49     STA  $49
D581-  A9 04     LDA  #$04
D583-  18        CLC             ; $0048/49: Startadresse des Eintrags
D584-  CA        DEX
D585-  F0 09     BEQ  $D590
D587-  6D 09 F0  ADC  $F009
D58A-  90 F7     BCC  $D583
D58C-  E6 49     INC  $49
D58E-  B0 F3     BCS  $D583

D590-  85 48     STA  $48
D592-  A0 13     LDY  #$13       ; BLOCKS USED und EOF werden auf den
D594-  B1 48     LDA  ($48)       neuen Stand gebracht:
D596-  79 8E EF  ADC  $EF8E,Y     BLOCKS USED plus 1
D599-  91 48     STA  ($48),Y     EOF plus $200
D59B-  C8        INY
D59C-  98        TYA
D59D-  49 18     EOR  #$18       ; anstelle von CMP (Carry!)
D59F-  D0 F3     BNE  $D594
D5A1-  20 DD DD  JSR  $DDDD      ; der Parent Block wird zurückgeschrieben
D5A4-  B0 0F     BCS  $D5B5
D5A6-  4C F7 D4  JMP  $D4F7      ; Kompletter Restart von CREATE
```

– Löscht den gesamten Directory Space

Xref: $D5B6 – CREATE Subdir Key Block oder (leeren) Datenblock des neuen Files

```
D5A9-  A9 00     LDA  #$00
D5AB-  AA        TAX
D5AC-  9D 00 F6  STA  $F600,X
D5AF-  9D 00 F7  STA  $F700,X
D5B2-  E8        INX
D5B3-  D0 F7     BNE  $D5AC
D5B5-  60        RTS
```

– Erstellung eines Subdirectory Key oder eines leeren Datenblocks
Erstellung eines File Info (für den File-Eintrag in der Parent Directory)

Xref: $D51D – JP von CREATE, wenn ein freier Directory-Eintrag gefunden wurde

```
D5B6-   20 A9 D5    JSR   $D5A9          ; Löscht den Directory Space
D5B9-   A0 0B       LDY   #$0B
D5BB-   B1 40       LDA   ($40),Y        ; CREATE DATE  &  TIME vom „user"
D5BD-   99 2F F0    STA   $F02F,Y
D5C0-   8A          TXA
D5C1-   11 40       ORA   ($40),Y
D5C3-   AA          TAX                  ; ORA für alle vier Bytes
D5C4-   88          DEY
D5C5-   C0 07       CPY   #$07
D5C7-   D0 F2       BNE   $D5BB
D5C9-   8A          TXA                  ; Resultat des ORA
D5CA-   D0 0B       BNE   $D5D7          ; DATE & TIME war vom „user" spezifiziert
D5CC-   A2 03       LDX   #$03
D5CE-   BD 90 BF    LDA   $BF90,X        ; ansonsten wird die System Time kopiert
D5D1-   9D 37 F0    STA   $F037,X
D5D4-   CA          DEX
D5D5-   10 F7       BPL   $D5CE
D5D7-   B1 40       LDA   ($40),Y        ; Storage Type vom „user"
D5D9-   C9 04       CMP   #$04
D5DB-   A9 10       LDA   #$10           ; 0(!)/1/2/3 => $10
D5DD-   90 02       BCC   $D5E1
D5DF-   A9 D0       LDA   #$D0           ; $D0: Subdirectory
D5E1-   AE 7A F0    LDX   $F07A          ; Start-Index des letzten Namens im Path
D5E4-   1D 00 F1    ORA   $F100,X          (von FIND FILE ENTRY übriggeblieben)
D5E7-   8D 1F F0    STA   $F01F          ; = Storage Type/Namenslänge
D5EA-   29 0F       AND   #$0F
D5EC-   A8          TAY                  ; Namenslänge
D5ED-   18          CLC
D5EE-   6D 7A F0    ADC   $F07A
D5F1-   AA          TAX
D5F2-   BD 00 F1    LDA   $F100,X
D5F5-   99 1F F0    STA   $F01F,Y        ; der Name wird auf $F020-$F02E kopiert
D5F8-   CA          DEX
D5F9-   88          DEY
D5FA-   D0 F6       BNE   $D5F2
D5FC-   A0 03       LDY   #$03
D5FE-   B1 40       LDA   ($40),Y
D600-   8D 3D F0    STA   $F03D          ; $F03D: ACCESS
D603-   C8          INY
D604-   B1 40       LDA   ($40),Y
D606-   8D 2F F0    STA   $F02F          ; $F02F: FILE TYPE
D609-   C8          INY
D60A-   B1 40       LDA   ($40),Y
D60C-   99 39 F0    STA   $F039,Y        ; $F03E/3F: AUX TYPE
D60F-   C0 06       CPY   #$06
D611-   D0 F6       BNE   $D609
D613-   AD B0 EF    LDA   $EFB0
D616-   8D 3B F0    STA   $F03B          ; $F03B/3C: ProDOS VERSION (00 00)
D619-   AD B1 EF    LDA   $EFB1
D61C-   8D 3C F0    STA   $F03C
D61F-   A9 01       LDA   #$01
D621-   8D 32 F0    STA   $F032          ; $F032/33: BLOCKS USED (00 01)
D624-   AD 1A F0    LDA   $F01A
D627-   8D 44 F0    STA   $F044          ; $F044/45: Nummer des Key Blocks der
D62A-   AD 1B F0    LDA   $F01B            Directory, in dem dieser File steht
D62D-   8D 45 F0    STA   $F045
```

– File Info ist erstellt: Subdirectory Key oder leerer Datenblock?

```
D630-   AD 1F F0    LDA   $F01F          ; Storage Type/Namenslänge
D633-   29 E0       AND   #$E0
D635-   F0 37       BEQ   $D66E          ; Sprung für Type $10 (Datenfile)
D637-   A2 1E       LDX   #$1E
D639-   BD 1F F0    LDA   $F01F,X
```

```
D63C-  9D 04 F6   STA  $F604,X      ; Name..CREATION DATE & TIME
D63F-  CA         DEX                 werden kopiert
D640-  10 F7      BPL  $D639
D642-  49 30      EOR  #$30         ; das letzte Byte ist $F01F:
D644-  8D 04 F6   STA  $F604        ; $D0 EOR $30 => $E0: Storage Type
D647-  A2 07      LDX  #$07           im Key Block
D649-  BD A6 EF   LDA  $EFA6,X      ; „uHUSTON"
D64C-  9D 14 F6   STA  $F614,X        ACCESS: $C3, EOF: $300, Eintragslänge im
D64F-  BD B0 EF   LDA  $EFB0,X        Parent: $27, Einträge pro Block $0D
D652-  9D 20 F6   STA  $F620,X      ; VERSION: 00 00
D655-  CA         DEX
D656-  10 F1      BPL  $D649
D658-  A2 02      LDX  #$02
D65A-  BD 1C F0   LDA  $F01C,X      ; Parent Pointer und Eintragsnummer
D65D-  9D 27 F6   STA  $F627,X        im Parent Directory Block
D660-  CA         DEX
D661-  10 F7      BPL  $D65A
D663-  AD 11 F0   LDA  $F011        ; Länge eines Eintrags im Parent
D666-  8D 2A F6   STA  $F62A
D669-  A9 02      LDA  #$02         ; EOF: $200 ($F034/36 = 00/00)
D66B-  8D 35 F0   STA  $F035
```

– Sprung hierher, wenn Datenfile: der Block enthält dann nur Nullen

```
D66E-  20 A9 DC   JSR  $DCA9        ; ALLOCATE NEXT BLOCK
D671-  B0 37      BCS  $D6AA
D673-  8D 30 F0   STA  $F030        ; Nummer des gefundenen freien Blocks
D676-  8C 31 F0   STY  $F031        ; Hibyte
D679-  85 46      STA  $46
D67B-  84 47      STY  $47
D67D-  20 DD DD   JSR  $DDDD        ; der eben erstellte Block wird
D680-  B0 28      BCS  $D6AA          geschrieben
D682-  EE 13 F0   INC  $F013        ; Nummer der aktiven DIR-Einträge => +1
D685-  D0 03      BNE  $D68A
D687-  EE 14 F0   INC  $F014
D68A-  20 AB D6   JSR  $D6AB        ; ENRTY & UPDATE, geht bis zur VolDIR
D68D-  B0 1B      BCS  $D6AA
D68F-  4C 86 DD   JMP  $DD86        ; Rewrite der VBM
```

– Berechnung der Speicheradresse eines Directory-Eintrags

Xref: $D6D9 – ENTRY & UPDATE (neuer Eintrag)
$D744 $- ENTRY & UPDATE (Loop – Finden des Eintrags im Parent)
$E78E – FLUSH (Zusammenstellen des File Info vor ENTRY & UPDATE)

```
D692-  A9 F6      LDA  #$F6         ; Directory-Block auf $F600
D694-  85 49      STA  $49
D696-  A9 04      LDA  #$04         ; Start mit $F604
D698-  AE 1E F0   LDX  $F01E        ; Eintragsnummer
D69B-  18         CLC
D69C-  CA         DEX
D69D-  F0 09      BEQ  $D6A8
D69F-  6D 11 F0   ADC  $F011        ; Länge eines Eintrags
D6A2-  90 F8      BCC  $D69C
D6A4-  E6 49      INC  $49
D6A6-  B0 F3      BCS  $D69B
D6A8-  85 48      STA  $48          ; $0048/49 werden gesetzt
D6AA-  60         RTS
```

**– ENTRY & UPDATE**
– Überschreiben eines Eintrags mit gesetzten Parametern

Xref: $D68A – CREATE (Datenfile/Subdirectory Key Block)
$E7CE – FLUSH (Update eines File-Eintrags)
JP $EA08 – SET FILE INFO (Überschreiben der File-Parameter – keine MOD TIME)
JP $EBDA – DESTROY (Setzt Eintrag ungültig: Namenslänge/Storage Type = 00)

```
D6AB-   AD 90 BF    LDA   $BF90
D6AE-   F0 0B       BEQ   $D6BB         ; keine System Time
D6B0-   A2 03       LDX   #$03
D6B2-   BD 90 BF    LDA   $BF90,X
D6B5-   9D 40 F0    STA   $F040,X       ; System Time wird kopiert
D6B8-   CA          DEX
D6B9-   10 F7       BPL   $D6B2
```

Xref: $EA05 JP – SET FILE INFO (MOD TIME angegeben)
$EB23 JP – RENAME (Überschreiben des File-Eintrags mit neuem Namen)

```
D6BB-   AD 3D F0    LDA   $F03D         ; ACCESS
D6BE-   0D 74 F0    ORA   $F074         ; internes BACKUP BIT, immer $20 (Ausnahme:
D6C1-   8D 3D F0    STA   $F03D                    SET F'INFO (s. „Spezielles")
D6C4-   AD 19 F0    LDA   $F019
D6C7-   8D 30 BF    STA   $BF30         ; Unitnummer
D6CA-   AD 1C F0    LDA   $F01C
D6CD-   85 46       STA   $46           ; Nummer des Directory-Blocks
D6CF-   AD 1D F0    LDA   $F01D
D6D2-   85 47       STA   $47
D6D4-   20 E1 DD    JSR   $DDE1         ; Lesen des Directory-Blocks auf $F600
D6D7-   B0 D1       BCS   $D6AA
D6D9-   20 92 D6    JSR   $D692         ; Berechnet die Eintrags-Startadresse
D6DC-   AC 11 F0    LDY   $F011         ; Länge eines Eintrags
D6DF-   88          DEY
D6E0-   B9 1F F0    LDA   $F01F,Y       ; der gesamte Eintrag wird kopiert
D6E3-   91 48       STA   ($48),Y
D6E5-   88          DEY
D6E6-   10 F8       BPL   $D6E0
D6E8-   AD 1A F0    LDA   $F01A         ; Nummer des Key Blocks der Directory
D6EB-   C5 46       CMP   $46           ; bereits im Key Block?
D6ED-   D0 07       BNE   $D6F6
D6EF-   AD 1B F0    LDA   $F01B
D6F2-   C5 47       CMP   $47
D6F4-   F0 14       BEQ   $D70A
D6F6-   20 DD DD    JSR   $DDDD         ; nicht im Key Block: der Block wird
D6F9-   B0 AF       BCS   $D6AA           zurückgeschrieben, der Key Block
D6FB-   AD 1A F0    LDA   $F01A
D6FE-   85 46       STA   $46
D700-   AD 1B F0    LDA   $F01B
D703-   85 47       STA   $47
D705-   20 E1 DD    JSR   $DDE1         ; wird geladen
D708-   B0 A0       BCS   $D6AA
D70A-   A0 01       LDY   #$01
D70C-   B9 13 F0    LDA   $F013,Y       ; die Anzahl der aktiven Einträge
D70F-   99 25 F6    STA   $F625,Y         wird kopiert
D712-   88          DEY                   (+1 für CREATE / -1 für DESTROY)
D713-   10 F7       BPL   $D70C
D715-   AD 10 F0    LDA   $F010         ; ACCESS der Directory – nie verändert
D718-   8D 22 F6    STA   $F622
```

```
D71B-  20 DD DD   JSR  $DDDD        ; der Key Block wird zurückgeschrieben
D71E-  B0 58      BCS  $D778
D720-  AD 04 F6   LDA  $F604
D723-  29 F0      AND  #$F0        ; war dieser Key Block der VolDIR Key?
D725-  49 F0      EOR  #$F0
D727-  F0 6D      BEQ  $D796       ; ja, RTS
D729-  AD 27 F6   LDA  $F627
D72C-  85 46      STA  $46
D72E-  AD 28 F6   LDA  $F628       ; der Parent Block dieser Directory wird
D731-  85 47      STA  $47           gesetzt,
D733-  AD 29 F6   LDA  $F629
D736-  8D 1E F0   STA  $F01E       ; die Eintragsnummer im Parent,
D739-  AD 2A F6   LDA  $F62A
D73C-  8D 11 F0   STA  $F011       ; und die Länge eines Eintrags im Parent.
D73F-  20 E1 DD   JSR  $DDE1       ; liest den Parent Block
D742-  B0 34      BCS  $D778
D744-  20 92 D6   JSR  $D692       ; setzt $0048/49 auf Start des Eintrags
D747-  AD 90 BF   LDA  $BF90
D74A-  F0 0D      BEQ  $D759       ; keine System Time
D74C-  A2 03      LDX  #$03
D74E-  A0 24      LDY  #$24
D750-  BD 90 BF   LDA  $BF90,X     ; Kopie der Sys Time in MOD DATE & TIME
D753-  91 48      STA  ($48),Y
D755-  88         DEY
D756-  CA         DEX
D757-  10 F7      BPL  $D750
D759-  20 DD DD   JSR  $DDDD       ; Schreibt den Block zurück
D75C-  B0 1A      BCS  $D778
D75E-  A0 25      LDY  #$25
D760-  B1 48      LDA  ($48),Y     ; ist dieser Block der Key Block der DIR?
D762-  C8         INY
D763-  C5 46      CMP  $46
D765-  85 46      STA  $46
D767-  D0 06      BNE  $D76F
D769-  B1 48      LDA  ($48),Y
D76B-  C5 47      CMP  $47
D76D-  F0 B1      BEQ  $D720       ; ja, Loop (Test auf VolDIR)
D76F-  B1 48      LDA  ($48),Y
D771-  85 47      STA  $47         ; Nein, Setzen des Key Blocks
D773-  20 E1 DD   JSR  $DDE1       ; und Lesen
D776-  90 A8      BCC  $D720       ; Loop (Test auf VolDIR)
D778-  60         RTS              ; RTS nur für Errors

D779-  A9 52      LDA  #$52        ; ERR#52: NOT A PRODOS VOLUME
D77B-  38         SEC
D77C-  60         RTS
```

– Prüfung, ob der geladene Key Block ein ProDOS (Vol)DIR Keyblock ist

Xref: $DACF – FIND FILE ENTRY (Absuchen aller Units)

```
D77D-  AD 00 F6   LDA  $F600       ; Backward Reference (lo)
D780-  CD AE EF   CMP  $EFAE       ; = 0
D783-  D0 F4      BNE  $D779
D785-  AD 01 F6   LDA  $F601       ; Backward Reference (hi)
D788-  CD AF EF   CMP  $EFAF       ; = 0
D78B-  D0 EC      BNE  $D779
D78D-  AD 04 F6   LDA  $F604
D790-  29 E0      AND  #$E0        ; muß Storage Type $F oder $E haben
D792-  C9 E0      CMP  #$E0
D794-  D0 E3      BNE  $D779
D796-  18         CLC
D797-  60         RTS
```

**– GET FILE**
(der File wird gesucht und die gefundenen Parameter werden kopiert)

Xref: $D339 – SET PREFIX (nicht PREFIX „/")
$E0B9 – OPEN
$E96F – GET FILE INFO
$E9D4 – SET FILE INFO
$EAB5 – RENAME (GET altes File)
$EB50 – DESTROY

```
D798-   20 AB D7    JSR   $D7AB         ; Sucht/lädt den DIR-Block, in dem der
D79B-   B0 0D       BCS   $D7AA           Eintrag steht
```

Xref: $E791 – FLUSH (DIR-Block bereits gefunden, die (alten) Parms werden kopiert)

```
D79D-   AC 11 F0    LDY   $F011         ; Länge eines Eintrags
D7A0-   B1 48       LDA   ($48),Y
D7A2-   99 1F F0    STA   $F01F,Y       ; der Eintrag wird kopiert
D7A5-   88          DEY
D7A6-   10 F8       BPL   $D7A0
D7A8-   A9 00       LDA   #$00          ; überflüssig, nirgends ausgewertet
D7AA-   60          RTS
```

**– FIND FILE ENTRY**
(Suche entlang des Pathname und Laden des Directory-Blocks)

Xref: $D4F7 – CREATE (Verifizierung, daß der neue File nicht existiert)
$D798 – GET FILE (s.o.)
$EA0B – RENAME (Verifizierung, daß der „alte" File existiert)
$EAA5 – RENAME (Verifizierung, daß das „neue" File nicht schon existiert)

– Suchen/Laden des entsprechenden DIR-Blocks

```
D7AB-   20 2F D9    JSR   $D92F         ; Suche des Volume und Lesen
D7AE-   B0 52       BCS   $D802           der Vol/Prefix DIR (Key Block)
D7B0-   D0 28       BNE   $D7DA         ; Sprung, wenn Pathname nicht zuende
```

– Pathname endet in einer Volume Directory
Wenn die VolDIR gelesen werden soll (z.B. OPEN + READ oder GET F'INFO), werden die Parameter, die in einer VolDIR nicht enthalten sind, schlicht von einer Tabelle kopiert:
Nummer des ersten Blocks: 00 02 / EOF $0800 / Blocks used: 00 04
Danach Exit mit ERR#40 (INVALID PATHNAME).
Bei CREATE/DESTROY führt dieser Error zum Abbruch, SET PREFIX/RENAME benutzen diese Fehlerbedingung zur Unterscheidung von Volume/Subdirectories.

```
D7B2-   AD 2A D8    LDA   $D82A         ; $F6
D7B5-   85 49       STA   $49
D7B7-   A9 04       LDA   #$04
D7B9-   85 48       STA   $48           ; $0048/49: $F604
D7BB-   A0 1F       LDY   #$1F
```

```
D7BD-  B1 48     LDA  ($48),Y
D7BF-  99 1F F0  STA  $F01F,Y      ; zuerst Kopie des gesamten Keys
D7C2-  88        DEY
D7C3-  C0 17     CPY  #$17
D7C5-  D0 F6     BNE  $D7BD
D7C7-  B9 A8 EF  LDA  $EFA8,Y      ; danach „Fälschen" der fehlenden
D7CA-  99 1F F0  STA  $F01F,Y        Parameter
D7CD-  88        DEY
D7CE-  C0 0F     CPY  #$0F
D7D0-  D0 F5     BNE  $D7C7
D7D2-  A9 D0     LDA  #$D0
D7D4-  8D 1F F0  STA  $F01F        ; Storage Type => $D (Subdirectory)
D7D7-  A9 40     LDA  #$40         ; ERR#40: INVALID PATHNAME
D7D9-  60        RTS               ; mit Carry gesetzt von „CPY #$0F"
```

– Suche entlang dem Pathname

```
D7DA-  A9 00     LDA  #$00         ; + Suche nach einem freien Dir-Eintrag
D7DC-  8D 5B F0  STA  $F05B          für CREATE: Bis jetzt keiner gefunden
D7DF-  38        SEC               ; Set: Überspringen des Keys in $D8D8
```

– Suche innerhalb einer Directory (mehrere Blöcke)

```
D7E0-  A9 00     LDA  #$00
D7E2-  8D 57 F0  STA  $F057        ; Zähler: Eintrag 00
D7E5-  20 D8 D8  JSR  $D8D8        ; Sucht Namen + freien Eintrag
D7E8-  90 65     BCC  $D84F        ; Name gefunden. Nächster Name im Path
D7EA-  AD 58 F0  LDA  $F058
D7ED-  ED 57 F0  SBC  $F057        ; nicht gefunden: die Anzahl der gefundenen
D7F0-  90 09     BCC  $D7FB          Einträge (Zähler $F057) in diesem Block
D7F2-  D0 10     BNE  $D804          wird von der Gesamtzahl der aktiven
D7F4-  CD 59 F0  CMP  $F059          Einträge abgezogen
D7F7-  F0 2A     BEQ  $D823        ; ok. Kein Eintrag mehr übrig => NOT FOUND
D7F9-  D0 09     BNE  $D804
D7FB-  CE 59 F0  DEC  $F059
D7FE-  10 04     BPL  $D804        ; noch Einträge übrig: nächster Block
D800-  A9 51     LDA  #$51         ; ERR#51: FILE STRUCTURE DAMAGED
D802-  38        SEC                 (mehr Einträge gezählt, als vorhanden
D803-  60        RTS                 sein sollten)
D804-  8D 58 F0  STA  $F058        ; Anzahl der übrigen aktiven Einträge
D807-  AD 2A D8  LDA  $D82A
D80A-  85 49     STA  $49          ; $0048/49 zurück auf $F604
D80C-  AD 02 F6  LDA  $F602
D80F-  D0 05     BNE  $D816        ; noch DIR-Blocks übrig
D811-  CD 03 F6  CMP  $F603
D814-  F0 EA     BEQ  $D800        ; keine Blocks mehr, aber noch Einträge!
D816-  85 46     STA  $46
D818-  AD 03 F6  LDA  $F603
D81B-  85 47     STA  $47
D81D-  20 E1 DD  JSR  $DDE1        ; Lesen des nächsten Blocks der Directory
D820-  90 BE     BCC  $D7E0        ; Loop
D822-  60        RTS
```

– Alle aktiven Einträge heruntergezählt. „Patch" für CREATE

```
D823-  AD 5B F0  LDA  $F05B        ; freier Eintrag gefunden (<>0)?
D826-  D0 1B     BNE  $D843
D828-  AD 02 F6  LDA  $F602        ; der letzte Block enthält 00 00
D82B-  D0 05     BNE  $D832          als Forward Reference
D82D-  CD 03 F6  CMP  $F603
D830-  F0 11     BEQ  $D843        ; war der letzte Block in dieser Dir
D832-  8D 1C F0  STA  $F01C        ; ansonsten muß der nächste Dir-Block
D835-  AD 03 F6  LDA  $F603          leer sein. $F01C/1D werden auf die
```

```
D838-  8D 1D F0    STA   $F01D          Blocknummer gesetzt,
D83B-  A9 01       LDA   #$01
D83D-  8D 1E F0    STA   $F01E        ; Nummer des „gefundenen" freien Eintrags
D840-  8D 5B F0    STA   $F05B        ; und Flag „freier Eintrag gefunden"
D843-  20 70 D9    JSR   $D970        ; Test: ist das der letzte Name im Path?
D846-  38          SEC
D847-  F0 03       BEQ   $D84C        ; Sprung, wenn am Ende des Path
D849-  A9 44       LDA   #$44         ; ERR#44: DIRECTORY NOT FOUND
D84B-  60          RTS
D84C-  A9 46       LDA   #$46         ; ERR#46: FILE NOT FOUND
D84E-  60          RTS
```

– Name im Directory gefunden

```
D84F-  20 69 D9    JSR   $D969        ; Index auf nächsten Namen im Path
D852-  F0 6C       BEQ   $D8C0        ; Path-Ende: Speichern der Blockno + Exit
D854-  A0 00       LDY   #$00
D856-  B1 48       LDA   ($48),Y      ; der Pathname ist nicht zuende: der
D858-  29 F0       AND   #$F0           gefundene Eintrag muß der Eintrag für
D85A-  C9 D0       CMP   #$D0           eine Subdirectory ($D) sein
D85C-  D0 E8       BNE   $D846        ; => ERR#46: DIRECTORY NOT FOUND
D85E-  A0 11       LDY   #$11         ; ist Subdirectory: die Nummer des
D860-  B1 48       LDA   ($48),Y        ersten („Key") Blocks wird auf $0046/47
D862-  85 46       STA   $46            und auf $F01A/1B („Key Blocknummer")
D864-  C8          INY                  kopiert.
D865-  8D 1A F0    STA   $F01A
D868-  B1 48       LDA   ($48),Y
D86A-  85 47       STA   $47
D86C-  8D 1B F0    STA   $F01B
D86F-  20 E1 DD    JSR   $DDE1        ; der Key Block wird gelesen
D872-  B0 24       BCS   $D898
D874-  AD 25 F6    LDA   $F625        ; Anzahl der aktiven Einträge =>
D877-  8D 58 F0    STA   $F058        ; Herunterzähler für den Such-Loop
D87A-  AD 26 F6    LDA   $F626
D87D-  8D 59 F0    STA   $F059
D880-  AD 21 F6    LDA   $F621        ; MLI VERSION: muß 00 sein, sonst
D883-  D0 11       BNE   $D896          ERR#4A: INCOMPATIBLE FILE FORMAT
D885-  AD 14 F6    LDA   $F614        ; $75 im Subdir Key (s. „Spezielles")
D888-  A2 00       LDX   #$00
D88A-  38          SEC
D88B-  2A          ROL
D88C-  90 01       BCC   $D88F        ; Zählung der gesetzten Bits
D88E-  E8          INX
D88F-  0A          ASL
D890-  D0 FA       BNE   $D88C
D892-  E0 05       CPX   #$05         ; $75 => 5 Bits gesetzt
D894-  F0 04       BEQ   $D89A
D896-  A9 4A       LDA   #$4A         ; ERR#4A: INCOMPATIBLE FILE FORMAT
D898-  38          SEC
D899-  60          RTS
D89A-  20 A0 D8    JSR   $D8A0        ; Kopiert die Parameter dieser DIR
D89D-  4C DA D7    JMP   $D7DA        ; Loop: nächster Name im Path
```

– Kopie der Parameter aus einem geladenen Key Block

Xref: $D89A – diese Directory ist Parent => Suche nach dem nächsten Namen im Path
$D944 – SEARCH VOLUME
$E774 – FLUSH: (Sub)dir gefunden, Kopie der Parms vor Suche des Filenamens

```
D8A0-  A2 0A     LDX  #$0A        ; DATE & TIME: $F00A..$F00D (nie benutzt)
D8A2-  BD 1C F6  LDA  $F61C,X       VERS: $F00E/0F  ACCESS: $F010
D8A5-  9D 0A F0  STA  $F00A,X       Länge eines Eintrags: $F011
D8A8-  CA        DEX                Anzahl Einträge im Block: $F012
D8A9-  10 F7     BPL  $D8A2         Gesamtzahl aktive Einträge: $F013/14
D8AB-  AD 04 F6  LDA  $F604       ; Storage Type
D8AE-  29 F0     AND  #$F0
D8B0-  49 F0     EOR  #$F0
D8B2-  F0 0B     BEQ  $D8BF       ; für VolDIR: RTS
D8B4-  A2 03     LDX  #$03        ; für Subdir:
D8B6-  BD 27 F6  LDA  $F627,X       Parent Pointer (Blockno Parent): $F006/07
D8B9-  9D 06 F0  STA  $F006,X       Eintragsnummer im  Parent-Block: $F008
D8BC-  CA        DEX                Länge eines Eintrags im Parent: $F009
D8BD-  10 F7     BPL  $D8B6
D8BF-  60        RTS
```

– Speichern der momentanen Eintragsnummer ($01-$0D) und Nummer des Dir-Blocks

Xref: $D8C0 – Name bei der Suche gefunden, Ende des Pathname
$D8F4 - Name im Dir-Block gefunden. (D8F4 vom FIND FILE-Loop aufgerufen).

```
D8C0-  AD 12 F0  LDA  $F012       ; Anzahl der Einträge per Block
D8C3-  38        SEC
D8C4-  ED 5A F0  SBC  $F05A       ; Herunterzähler vom Such-Loop
D8C7-  69 00     ADC  #$00        ; + 1 (Carry set) => Bereich $01-$0D
D8C9-  8D 1E F0  STA  $F01E       ; Nummer des momentanen Eintrags
D8CC-  A5 46     LDA  $46
D8CE-  8D 1C F0  STA  $F01C       ; Nummer des Directory-Blocks, in dem
D8D1-  A5 47     LDA  $47           der Eintrag gefunden wurde
D8D3-  8D 1D F0  STA  $F01D
D8D6-  18        CLC
D8D7-  60        RTS
```

– Suche nach einem Namen des Path innerhalb eines Dir-Blocks

Xref: $D7E5 – Absuchen einer Directory in FIND FILE

```
D8D8-  AD 12 F0  LDA  $F012       ; Anzahl der Einträge per Dir-Block
D8DB-  8D 5A F0  STA  $F05A       ; Herunterzähler
D8DE-  AD 2A D8  LDA  $D82A
D8E1-  85 49     STA  $49
D8E3-  A9 04     LDA  #$04
D8E5-  85 48     STA  $48         ; $0048/49 = $F604
D8E7-  B0 33     BCS  $D91C       ; Aufruf mit SEC: Erster Eintrag („Key")
D8E9-  A0 00     LDY  #$00                          wird übersprungen
D8EB-  B1 48     LDA  ($48),Y     ; Typ/Längenbyte eines Eintrags
D8ED-  D0 0D     BNE  $D8FC       ; Eintrag ist besetzt: Sprung
D8EF-  AD 5B F0  LDA  $F05B       ; „Freier Eintrag gefunden"-Flag
D8F2-  D0 28     BNE  $D91C       ; Sprung, wenn bereits vorher gesetzt
D8F4-  20 C0 D8  JSR  $D8C0       ; Ansonsten Speichern von Blockno/Eintragsno
D8F7-  EE 5B F0  INC  $F05B         und Setzen von „Freier Eintrag gefunden"
D8FA-  D0 20     BNE  $D91C       ; „always": NEXT (nächster Eintrag)

D8FC-  29 0F     AND  #$0F        ; Sprung hierher, wenn Eintrag besetzt
D8FE-  EE 57 F0  INC  $F057       ; Zähler der „aktiven" Einträge
D901-  8D 78 F0  STA  $F078       ; Länge des Namens: Zähler für Vergleich
D904-  AE 7A F0  LDX  $F07A       ; Index im Pathname
D907-  DD 00 F1  CMP  $F100,X     ; dieselbe Länge?
D90A-  D0 10     BNE  $D91C       ; nein, Namen müssen verschieden sein: NE
D90C-  E8        INX
```

```
D90D-   C8          INY
D90E-   B1 48       LDA   ($48),Y       ; Vergleich mit dem Namen im Path
D910-   DD 00 F1    CMP   $F100,X
D913-   D0 07       BNE   $D91C
D915-   CE 78 F0    DEC   $F078
D918-   D0 F2       BNE   $D90C         ; bis herunter zum ersten Zeichen
D91A-   18          CLC
D91B-   60          RTS                 ; Name bzw. Eintrag gefunden

D91C-   CE 5A F0    DEC   $F05A         ; Loop-Zähler
D91F-   38          SEC
D920-   F0 F9       BEQ   $D91B         ; Name nicht gefunden
D922-   AD 11 F0    LDA   $F011         ; Länge eines Eintrags
D925-   18          CLC
D926-   65 48       ADC   $48           ; $0048/49 auf nächsten Eintrag
D928-   90 BB       BCC   $D8E5
D92A-   E6 49       INC   $49
D92C-   18          CLC
D92D-   90 B6       BCC   $D8E5         ; Loop: nächster Name
```

**– GET VOLDIR/PREFIX KEY BLOCK**

– Setzen der DIR-Parameter

Xref: $D7AB – FIND FILE ENTRY

```
D92F-   20 88 D9    JSR   $D988         ; GET VOLUME sucht & lädt VolDIR Key Block
D932-   B0 53       BCS   $D987
D934-   A9 00       LDA   #$00          ; Volume gefunden:
D936-   A0 42       LDY   #$42
D938-   99 06 F0    STA   $F006,Y       ; $F006..$F048 wird gelöscht
D93B-   88          DEY
D93C-   10 FA       BPL   $D938
D93E-   AD 30 BF    LDA   $BF30         ; Unitnummer, gesetzt von GET VOLUME
D941-   8D 19 F0    STA   $F019
D944-   20 A0 D8    JSR   $D8A0         ; Kopiert DATE & TIME/ACCESS.. aus Key
D947-   A0 01       LDY   #$01            Block, bei Subdir auch Parent Pointer
D949-   AE 51 F0    LDX   $F051         ; VCB-Index, gesetzt von GET VOLUME
D94C-   E8          INX
D94D-   BD 12 F2    LDA   $F212,X       ; TOTAL BLOCKS auf dem Volume ($F212) und
D950-   99 17 F0    STA   $F017,Y         Blocknummer des VolDIR Key Blocks (F21A)
D953-   BD 1A F2    LDA   $F21A,X         müssen aus dem VCB kopiert werden, eine
D956-   99 15 F0    STA   $F015,Y         PREFIXed Subdirectory enthält diese
D959-   B9 46 00    LDA   $0046,Y         Informationen nicht!
D95C-   99 1A F0    STA   $F01A,Y       ; Blocknummer dieses Keys (f. SubDIRs)
D95F-   B9 13 F0    LDA   $F013,Y       ; Anzahl der aktiven Einträge als
D962-   99 58 F0    STA   $F058,Y       ; Herunterzähler für die Suche nach dem
D965-   CA          DEX                   nächsten Namen (= Eintrag) im Path
D966-   88          DEY
D967-   10 E4       BPL   $D94D
```

– Überspringen des momentanen Namen im Path

Xref: $D84F – Name in Directory gefunden

```
D969-   20 70 D9    JSR   $D970         ; berechnet den Index des nächsten Namens
D96C-   8C 7A F0    STY   $F07A         ; Index wird gesetzt
D96F-   60          RTS                 ; Z-Flag gesetzt bei Namensende (Index = 00)
```

– Berechnung des Index des nächsten Namens im Path

Xref: $D843 – Test auf Ende des Pathname (=> DIR oder FILE NOT FOUND)
$D96C – Überspringen („Abhaken") des momentanen Namens im Path

```
D970-  AC 7A F0    LDY   $F07A        ; Startindex des momentanen Namens
D973-  B9 00 F1    LDA   $F100,Y      ; = Längenbyte
D976-  38          SEC                ; Name + Längenbyte = Namenslänge + 1
D977-  6D 7A F0    ADC   $F07A        ; => 00 bei Namensende
D97A-  A8          TAY
D97B-  D0 06       BNE   $D983        ; nicht zuende
D97D-  AD 30 BF    LDA   $BF30
D980-  8D 5F F0    STA   $F05F        ; Namensende: Unitnummer wird gespeichert
D983-  B9 00 F1    LDA   $F100,Y      ; Länge des nächsten Namens
D986-  18          CLC
D987-  60          RTS
```

**– SEARCH VOLUME/PREFIXED DIRECTORY**

Xref: $D92F – GET VOLUME/PREFIXED DIRECTORY KEY BLOCK

```
D988-  A9 00       LDA   #$00
D98A-  AC 9A BF    LDY   $BF9A        ; System PREFIX
D98D-  2C 7C F0    BIT   $F07C        ; FULL NAME Flag
D990-  10 01       BPL   $D993        ; Name nicht vollständig
D992-  A8          TAY
D993-  8C 7B F0    STY   $F07B        ; 00 (voller Name) oder Start SYS PREFIX
D996-  8D 30 BF    STA   $BF30        ; = 00
```

– Absuchen des VCB nach dem Volume-Namen

```
D999-  48          PHA                ; Loop im VCB: Start mit Index 00
D99A-  AA          TAX
D99B-  BD 00 F2    LDA   $F200,X      ; Namenslänge eines VCB-Eintrags
D99E-  D0 0C       BNE   $D9AC        ; Eintrag enthält gültige Daten
D9A0-  AC 7B F0    LDY   $F07B        ; VCB-Eintrag unbesetzt: NE
D9A3-  68          PLA
D9A4-  18          CLC
D9A5-  69 20       ADC   #$20         ; nächster Index
D9A7-  90 F0       BCC   $D999        ; nächster Eintrag
D9A9-  4C FD D9    JMP   $D9FD        ; nicht gefunden

D9AC-  8D 78 F0    STA   $F078        ; Namenslänge: Herunterzähler
D9AF-  D9 00 F1    CMP   $F100,Y      ; Name im Path (Start mit Namenslänge)
D9B2-  D0 EC       BNE   $D9A0        ; ungleich: NE
D9B4-  E8          INX
D9B5-  C8          INY
D9B6-  BD 00 F2    LDA   $F200,X
D9B9-  CE 78 F0    DEC   $F078
D9BC-  10 F1       BPL   $D9AF
D9BE-  68          PLA                ; Namen sind gleich (incl. Länge)
D9BF-  AA          TAX                ; = Startindex im VCB für diesen Eintrag
D9C0-  8D 51 F0    STA   $F051        ; VCB-Index wird gesetzt
D9C3-  BD 10 F2    LDA   $F210,X
D9C6-  8D 30 BF    STA   $BF30        ; Unitnummer für diesen Eintrag
D9C9-  A9 00       LDA   #$00
D9CB-  85 47       STA   $47
D9CD-  A9 02       LDA   #$02         ; Blocknummer 00 02 = VolDIR Key Block
D9CF-  85 46       STA   $46
D9D1-  AD 7B F0    LDA   $F07B        ; FULL NAME Flag
```

```
D9D4-  A8          TAY
D9D5-  8D 7A F0    STA   $F07A
D9D8-  F0 10       BEQ   $D9EA        ; 00: Full Name, VolDIR Key wird gelesen
D9DA-  38          SEC                ; <> 00 => SYS PREFIX gesezt, $F060/61 =>
D9DB-  79 00 F1    ADC   $F100,Y        Blockno PREFIX DIR. Der Index im Pathname
D9DE-  90 F4       BCC   $D9D4          wird solange erhöht, bis der letzte
D9E0-  AD 60 F0    LDA   $F060          Name im PREFIX erreicht ist. Die PREFIX
D9E3-  85 46       STA   $46            DIR muß ebenfalls diesen Namen haben
D9E5-  AD 61 F0    LDA   $F061
D9E8-  85 47       STA   $47          ; Setzen der PREFIX Blocknummer

D9EA-  20 E1 DD    JSR   $DDE1        ; der Key der (PREFIX/VOL)DIR wird gelesen
D9ED-  B0 06       BCS   $D9F5
D9EF-  20 91 DA    JSR   $DA91        ; Test auf ProDOS-Directory/Namensvergleich
D9F2-  B0 01       BCS   $D9F5        ; Stimmt nicht überein (Disk gewechselt)
D9F4-  60          RTS                ; ok. (PREFIX) DIR gefunden & Key geladen
```

– Name stimmt nicht überein (Vol/PREFIX DIR)

Xref: $D9F2 – Disk wurde gewechselt oder Lesefehler

```
D9F5-  AE 51 F0    LDX   $F051        ; VCB-Index
D9F8-  BD 11 F2    LDA   $F211,X      ; OPEN File auf diesem Volume?
D9FB-  30 19       BMI   $DA16        ; Ja, es darf nichts im VCB geändert werden!
```

Xref: $D9A9 – Volume im nicht im VCB gefunden

```
D9FD-  AD 7B F0    LDA   $F07B        ; Reload des Pathname-Starts
DA00-  8D 7A F0    STA   $F07A
DA03-  20 57 DA    JSR   $DA57        ; kopiert alle „connected" Units auf $F08A...
DA06-  AD 30 BF    LDA   $BF30        ; wurde bereits versucht, von einem Volume
DA09-  D0 0F       BNE   $DA1A          zu lesen (Disk gewechselt)?. Wenn ja,
                                        dann Start mit diesem Unit
DA0B-  AE 31 BF    LDX   $BF31        ; ansonsten Start von „oben herunter"
DA0E-  BD 8A F0    LDA   $F08A,X      ; von $DA57 kopierte DEVICE TABLE
DA11-  D0 0A       BNE   $DA1D
DA13-  CA          DEX                ; Unit bereits erledigt (s. $DA28)
DA14-  10 F8       BPL   $DA0E        ; Loop bis zum letzten („untersten") Unit
DA16-  A9 45       LDA   #$45         ; ERR=45: VOLUME NOT FOUND
DA18-  38          SEC
DA19-  60          RTS

DA1A-  AE 31 BF    LDX   $BF31        ; Einsprung von $DA09

DA1D-  DD 8A F0    CMP   $F08A,X
DA20-  F0 03       BEQ   $DA25        ; wenn via $DA09: Suche nach dem zuletzt
DA22-  CA          DEX                  benutzten Unit
DA23-  10 F8       BPL   $DA1D

DA25-  8D 30 BF    STA   $BF30        ; die Unitnummer wird gesetzt
DA28-  A9 00       LDA   #$00
DA2A-  9D 8A F0    STA   $F08A,X      ; und das DEVICE aus der Tabelle gelöscht
DA2D-  20 69 DA    JSR   $DA69        ; Suche nach einem freien Eintrag oder
DA30-  B0 24       BCS   $DA56          einem Eintrag mit dieser Unitno
DA32-  AE 51 F0    LDX   $F051        ; von $DA69 gesetzter Index
DA35-  BD 00 F2    LDA   $F200,X
DA38-  F0 05       BEQ   $DA3F        ; Sprung, wenn der Eintrag frei ist
DA3A-  BD 11 F2    LDA   $F211,X      ; Eintrag hat dieselbe Unitno. OPEN File?
DA3D-  30 CC       BMI   $DA0B        ; Wenn OPEN File, nächstes Unit

DA3F-  A9 00       LDA   #$00         ; freier VCB-Eintrag oder kein OPEN File:
DA41-  85 47       STA   $47            Block 00 02 (VolDIR Key) wird gesetzt
```

```
DA43-  A9 02     LDA  #$02
DA45-  85 46     STA  $46
DA47-  20 E1 DD  JSR  $DDE1      ; und gelesen
DA4A-  B0 BF     BCS  $DA0B
DA4C-  20 B7 DA  JSR  $DAB7      ; erstellt den VCB-Eintrag mit den Daten
DA4F-  B0 BA     BCS  $DA0B        des gelesenen Key Blocks/Test ProDOS DIR
DA51-  20 91 DA  JSR  $DA91      ; Vergleich Key/Name im Path
DA54-  B0 B5     BCS  $DA0B      ; nicht gleich: Loop
DA56-  60        RTS             ; ok. Gesuchtes Volume gefunden
```

– Erstellt eine Kopie aller „connected" Units von der System Global Page

Xref: $D461 – ONLINE ALL
$DA03 – GET VOLUME – Absuchen aller Units nach einem bestimmten Volume

```
DA57-  AE 31 BF  LDX  $BF31      ; DEVNUM - Anzahl der Units
DA5A-  BD 32 BF  LDA  $BF32,X    ; DEVICE TABLE auf der SGP
DA5D-  29 F0     AND  #$F0       ; Ausblenden der „Characteristics",
DA5F-  9D 8A F0  STA  $F08A,X      übrig bleibt DSSS (Drive/Slot)
DA62-  CA        DEX
DA63-  10 F5     BPL  $DA5A
DA65-  AE 31 BF  LDX  $BF31
DA68-  60        RTS
```

– Sucht nach einem (freien) VCB-Eintrag und setzt den VCB-Index
Aufruf mit gesetzter Unitnummer. Wird kein Eintrag für diese Unitno gefunden, dann Suche nach einem freien Eintrag. Möglicher Fehler: VCB TABLE FULL.

Xref: $D483 – ONLINE SPECIFIC
$DA2D – GET VOLUME – Volume in VCB nicht gefunden, internes ONLINE ALL
$E6D0 – CLOSE single file: Suche nach dem VCB-Eintrag, um die Anzahl der OPEN Files um eins zu erniedrigen

```
DA69-  A9 00     LDA  #$00
DA6B-  A0 FF     LDY  #$FF       ; Annahme: keine freien Einträge
DA6D-  AA        TAX
DA6E-  BD 10 F2  LDA  $F210,X    ; zugeordnete Unitnummer
DA71-  CD 30 BF  CMP  $BF30      ; ist die gesuchte Unitno?
DA74-  D0 05     BNE  $DA7B
DA76-  8E 51 F0  STX  $F051      ; ja, VCB-Index wird gesetzt
DA79-  18        CLC
DA7A-  60        RTS
DA7B-  BD 00 F2  LDA  $F200,X    ; Ist der Eintrag frei?
DA7E-  D0 04     BNE  $DA84      ; nein: NE
DA80-  C8        INY             ; ja: freier Eintrag gefunden (Y <> $FF)
DA81-  8E 51 F0  STX  $F051      ; FCB-Index wird gesetzt
DA84-  8A        TXA             ; nächster Eintrag (NEXT)
DA85-  18        CLC
DA86-  69 20     ADC  #$20
DA88-  D0 E3     BNE  $DA6D
DA8A-  98        TYA
DA8B-  30 01     BMI  $DA8E      ; kein freier Eintrag gefunden (Y = $FF)
DA8D-  18        CLC
DA8E-  A9 55     LDA  #$55       ; ERR#55: VCB TABLE FULL
DA90-  60        RTS
```

– Prüft den geladenen Block auf ProDOS (Sub)DIR Key & Namensvergleich

Xref: $D9EF – FIND FILE – Volume oder PREFIX Dir gefunden
$DA51 – GET VOLUME – Volume nicht im VCB gefunden, internes ONLINE ALL

```
DA91-  A2 00      LDX   #$00
DA93-  AC 7A F0   LDY   $F07A        ; Startindex des momentanen Namens im Path
DA96-  AD 04 F6   LDA   $F604        ; Storage Type
DA99-  C9 E0      CMP   #$E0         ; muß $E oder $F sein
DA9B-  90 07      BCC   $DAA4
DA9D-  29 0F      AND   #$0F         ; => Namenslänge
DA9F-  8D 78 F0   STA   $F078        ; Herunterzähler
DAA2-  D0 05      BNE   $DAA9        ; muß > 0 sein
DAA4-  38         SEC
DAA5-  60         RTS
DAA6-  BD 04 F6   LDA   $F604,X      ; Vergleich Key - Name im Path
DAA9-  D9 00 F1   CMP   $F100,Y
DAAC-  D0 F6      BNE   $DAA4
DAAE-  E8         INX
DAAF-  C8         INY
DAB0-  CE 78 F0   DEC   $F078
DAB3-  10 F1      BPL   $DAA6
DAB5-  18         CLC                ; ok. Namen sind gleich
DAB6-  60         RTS
```

– Kopiert die Parameter der gelesenen VolDIR in den VCB

Xref: $DA4C – GET VOLUME – Volume nicht in VCB gefunden, internes ONLINE ALL

```
DAB7-  AE 51 F0   LDX   $F051        ; VCB-Index, gesetzt von $DA69
DABA-  BD 00 F2   LDA   $F200,X
DABD-  F0 05      BEQ   $DAC4        ; war ein freier Eintrag
DABF-  20 1C DB   JSR   $DB1C        ; ansonsten Vergleich VCB - Key
DAC2-  90 57      BCC   $DB1B        ; sind gleich - kein Update
```

Xref: $D4A2 – ONLINE: der alte VCB-Eintrag ist leer oder hat keinen OPEN File

```
DAC4-  A9 00      LDA   #$00
DAC6-  A0 1F      LDY   #$1F         ; der gesamte VCB-Eintrag wird gelöscht
DAC8-  9D 00 F2   STA   $F200,X
DACB-  E8         INX
DACC-  88         DEY
DACD-  10 F9      BPL   $DAC8

DACF-  20 7D D7   JSR   $D77D        ; Prüfung auf ProDOS VolDIR Key Block
DAD2-  B0 47      BCS   $DB1B        ; kein ProDOS-Volume
DAD4-  20 3D DB   JSR   $DB3D        ; Absuchen des VCB - existiert der NAME(!)?
DAD7-  B0 41      BCS   $DB1A          CLC/RTS wenn der Volume-Name mit einem
DAD9-  AD 04 F6   LDA   $F604          OPEN File existiert. $F075/76 sind dann
DADC-  29 0F      AND   #$0F           auf Flag/VCB-Index gesetzt
DADE-  A8         TAY                ; Namenslänge
DADF-  48         PHA
DAE0-  0D 51 F0   ORA   $F051        ; VCB-Index + Namenslänge
DAE3-  AA         TAX
DAE4-  B9 04 F6   LDA   $F604,Y
DAE7-  9D 00 F2   STA   $F200,X      ; Kopie des Volume-Namens in den VCB-Eintrag
DAEA-  CA         DEX
```

```
DAEB-  88          DEY
DAEC-  D0 F6       BNE   $DAE4
DAEE-  68          PLA                ; Namenslänge
DAEF-  9D 00 F2    STA   $F200,X      ; OHNE Storage Type
DAF2-  AD 30 BF    LDA   $BF30
DAF5-  9D 10 F2    STA   $F210,X      ; Byte $10: Unitnummer
DAF8-  AD 29 F6    LDA   $F629
DAFB-  9D 12 F2    STA   $F212,X      ; Byte $12/$13: TOTAL BLOCKS
DAFE-  AD 2A F6    LDA   $F62A
DB01-  9D 13 F2    STA   $F213,X
DB04-  A5 46       LDA   $46
DB06-  9D 16 F2    STA   $F216,X      ; Bytes $16/$17: Blocknummer des VolDIR Keys
DB09-  A5 47       LDA   $47
DB0B-  9D 17 F2    STA   $F217,X
DB0E-  AD 27 F6    LDA   $F627
DB11-  9D 1A F2    STA   $F21A,X      ; Bytes $1A/$1B: Blocknummer VBM
DB14-  AD 28 F6    LDA   $F628                        (erster VBM-Block)
DB17-  9D 1B F2    STA   $F21B,X
DB1A-  18          CLC
DB1B-  60          RTS
```

– Vergleicht die neu geladene VolDIR mit dem alten VCB-Eintrag

Xref: $D4B1 – ONLINE ALL (nur sinnvoll für Volumes mit OPEN File)
$DABF – Kopieren der gelesenen VolDIR in den VCB
$DB40 – Absuchen des gesamten VCB auf Namensbasis

```
DB1C-  AD 04 F6    LDA   $F604        ; Storage Type/Namenslänge
DB1F-  29 0F       AND   #$0F         ; Namenslänge
DB21-  DD 00 F2    CMP   $F200,X      ; vs. Name im VCB-Eintrag
DB24-  D0 0E       BNE   $DB34        ; SEC/RTS
DB26-  A8          TAY
DB27-  8E 50 F0    STX   $F050        ; Scratch
DB2A-  0D 50 F0    ORA   $F050
DB2D-  AA          TAX
DB2E-  B9 04 F6    LDA   $F604,Y      ; Vergleich des gesamten Namens von hinten
DB31-  DD 00 F2    CMP   $F200,X
DB34-  38          SEC                ; RTS mit SEC, wenn Namen ungleich
DB35-  D0 05       BNE   $DB3C
DB37-  CA          DEX
DB38-  88          DEY
DB39-  D0 F3       BNE   $DB2E
DB3B-  18          CLC                ; ok. Namen sind gleich
DB3C-  60          RTS
```

– Absuchen des gesamten VCB auf Namensbasis

Xref: $DAD4 – Update des VCB mit geladener VolDIR

```
DB3D-  A9 00       LDA   #$00         ; Start mit erstem VCB-Eintrag (Index)
DB3F-  AA          TAX
DB40-  20 1C DB    JSR   $DB1C        ; Namensvergleich Eintrag – Key Block
DB43-  B0 0F       BCS   $DB54        ; ungleich: NE
DB45-  BD 11 F2    LDA   $F211,X      ; OPEN File im VCB-Eintrag?
DB48-  30 14       BMI   $DB5E        ; ja, Setzen von $F075/76 + Exit
DB4A-  A9 00       LDA   #$00
DB4C-  9D 00 F2    STA   $F200,X      ; ansonsten wird der VCB-Eintrag
DB4F-  9D 10 F2    STA   $F210,X        ungültig gemacht
DB52-  F0 08       BEQ   $DB5C        ; „always"
DB54-  8A          TXA                ; nächster VCB-Eintrag
DB55-  18          CLC
```

```
DB56-  29 E0      AND   #$E0          ; X-Reg ist auch Index im Namen bei $DB1C,
DB58-  69 20      ADC   #$20            zurück auf die Basisadresse
DB5A-  90 E3      BCC   $DB3F         ; nächster VCB-Eintrag
DB5C-  18         CLC                 ; ok. Nicht gefunden bzw. kein OPEN File
DB5D-  60         RTS

DB5E-  8D 75 F0   STA   $F075         ; OPEN File im „alten" VCB-Eintrag, die
DB61-  8E 76 F0   STX   $F076           Namen sind gleich, die Unitnummern
DB64-  38         SEC                 ; stimmen aber nicht überein!
DB65-  60         RTS
```

**– SPACE ON VOLUME?**

(Berechnung der freien Blocks auf einem Volume)

Beim „Login" eines Volume wird nur TOTAL BLOCKS im VCB gesetzt, die Anzahl der freien Blocks muß erst berechnet werden.

Xref: $E4BE – WRITE – nach der Bestimmung der benötigten Blocks via SET MARK
$EB69 – DESTROY – Aufruf mit „Blocks needed" = 0

```
DB66-  AE 51 F0   LDX   $F051         ; VCB-Index: Wenn die Berechnung bereits
DB69-  BD 15 F2   LDA   $F215,X         seit dem „Login" durchgeführt wurde, sind
DB6C-  1D 14 F2   ORA   $F214,X         die Bytes $14/$15 im VCB-Eintrag <> 0
DB6F-  D0 52      BNE   $DBC3          gesetzt (Ausnahme: 00 Blocks frei).
```

Xref: $E989 – GET F'INFO für VolDIR – BLOCKS USED (TOTAL minus BLOCKS FREE)

```
DB71-  20 15 DC   JSR   $DC15         ; Berechnet die Maximalzahl der VBM-Blocks
DB74-  8D 5C F0   STA   $F05C           auf diesem Volume aus TOTAL BLOCKS
DB77-  A9 00      LDA   #$00
DB79-  8D 46 F0   STA   $F046         ; $F046/47: Zähler für freie Blocks
DB7C-  8D 47 F0   STA   $F047
DB7F-  A9 FF      LDA   #$FF
DB81-  8D 5B F0   STA   $F05B         ; „no Space"- Flag und VBM „Extent"
DB84-  20 86 DD   JSR   $DD86         ; REWRITE VBM – falls sich noch irgendein
DB87-  B0 4E      BCS   $DBD7           geänderter VBM-Block im Speicher befindet
DB89-  AE 51 F0   LDX   $F051         ; VCB-Index
DB8C-  BD 1A F2   LDA   $F21A,X
DB8F-  85 46      STA   $46           ; die Nummer des (ersten) VBM-Blocks
DB91-  BD 1B F2   LDA   $F21B,X         wird gesetzt
DB94-  85 47      STA   $47
```

– Loop durch alle möglichen VBM-Blocks auf diesem Volume

```
DB96-  20 E1 DD   JSR   $DDE1         ; liest einen VBM-Block auf $F600(!)
DB99-  B0 3C      BCS   $DBD7
DB9B-  20 D8 DB   JSR   $DBD8         ; Anzahl der freien Blocks => $F046/47
DB9E-  CE 5C F0   DEC   $F05C         ; Herunterzähler (s. $DB74)
DBA1-  30 09      BMI   $DBAC         ; keine weiteren VBM-Blocks
DBA3-  E6 46      INC   $46
DBA5-  D0 EF      BNE   $DB96         ; ansonsten nächster VBM-Block: Loop
DBA7-  E6 47      INC   $47
DBA9-  4C 96 DB   JMP   $DB96

DBAC-  AE 51 F0   LDX   $F051
DBAF-  AD 5B F0   LDA   $F05B         ; mindestens ein freier Block gefunden?
DBB2-  30 20      BMI   $DBD4         ; nein – ERR#48
DBB4-  9D 1C F2   STA   $F21C,X       ; „VBM extent", auf dem ein freier Block ge-
```

```
DBB7-   AD 47 F0    LDA   $F047          funden wurde, $F05B von $DBFE gesetzt
DBBA-   9D 15 F2    STA   $F215,X      ; Anzahl der freien Blocks
DBBD-   AD 46 F0    LDA   $F046
DBC0-   9D 14 F2    STA   $F214,X
```

– Bestimmung der freien Blocks ist erfolgt

```
DBC3-   BD 14 F2    LDA   $F214,X      ; Anzahl der freien Blocks
DBC6-   38          SEC
DBC7-   ED 54 F0    SBC   $F054        ; BLOCKS NEEDED (low)
DBCA-   BD 15 F2    LDA   $F215,X
DBCD-   ED 55 F0    SBC   $F055        ; hi – sehr gründlich & nie benutzt (00)
DBD0-   90 02       BCC   $DBD4        ; VOLUME FULL
DBD2-   18          CLC
DBD3-   60          RTS

DBD4-   A9 48       LDA   #$48         ; ERR=48: VOLUME FULL
DBD6-   38          SEC
DBD7-   60          RTS
```

– CALCFREE
– Berechnung der freien Blocks in einem VBM-Block, setzt $F05B = Extentno.

Xref: $DB9B – Berechnung sämtlicher freien Blocks auf einem Volume

```
DBD8-   A0 00       LDY   #$00
DBDA-   B9 00 F6    LDA   $F600,Y
DBDD-   F0 03       BEQ   $DBE2        ; alle 8 Blocks belegt
DBDF-   20 05 DC    JSR   $DC05        ; zählt die gesetzten Bits im Byte
DBE2-   B9 00 F7    LDA   $F700,Y
DBE5-   F0 03       BEQ   $DBEA        ; „obere Hälfte" des VBM-Blocks
DBE7-   20 05 DC    JSR   $DC05
DBEA-   C8          INY
DBEB-   D0 ED       BNE   $DBDA
DBED-   2C 5B F0    BIT   $F05B        ; „No Space"-Flag bereits zurückgesetzt?
DBF0-   10 12       BPL   $DC04        ; RTS wenn ja
DBF2-   AD 46 F0    LDA   $F046
DBF5-   0D 47 F0    ORA   $F047
DBF8-   F0 0A       BEQ   $DC04        ; RTS wenn 00 00 Blocks frei
DBFA-   20 15 DC    JSR   $DC15        ; Berechnung der max. VBM-Blocks
DBFD-   38          SEC
DBFE-   ED 5C F0    SBC   $F05C        ; minus Herunterzähler
DC01-   8D 5B F0    STA   $F05B        ; = erster VBM-Block mit freien Blocks
DC04-   60          RTS                              (Extent No., d.h. 0,1,2..)
```

– Zählung der gesetzten Bits im Akku, Erhöhung von $F046/47

Xref: $DBOF – CALCFREE Loop für untere Hälfte eines VBM-Blocks
$DBDF – CALCFREE loop für obere Hälfte eines VBM-Blocks

```
DC05-   0A          ASL
DC06-   90 08       BCC   $DC10
DC08-   EE 46 F0    INC   $F046        ; Bit gesetzt: Erhöhung des
DC0B-   D0 03       BNE   $DC10          „Free Space Count"
DC0D-   EE 47 F0    INC   $F047
DC10-   09 00       ORA   #$00         ; Loop, solange noch Bits gesetzt sind
DC12-   D0 F1       BNE   $DC05
DC14-   60          RTS
```

– Berechnung der maximal möglichen Zahl der VBM-Blocks -1 für ein Volume

Xref: $DB71 – SPACE ON VOL – Berechnung des Herunterzählers für den Loop
$DBFA – CALCFREE – Setzen des ersten VBM-Extent mit freien Blocks (=> $F05B)

```
DC15-  AE 51 F0    LDX   $F051         ; VCB-Index
DC18-  BC 13 F2    LDY   $F213,X       ; TOTAL BLOCKS (hi)
DC1B-  BD 12 F2    LDA   $F212,X       ; lo - geben zusammen die Gesamtzahl
DC1E-  D0 01       BNE   $DC21           der Blocks auf dem Volume an.
DC20-  88          DEY
DC21-  98          TYA                   z.B.: $10 00 Blocks => 00, aber
DC22-  4A          LSR                         $10 01 Blocks => 01, d.h. es
DC23-  4A          LSR                   existieren dann 2 VBM-Blocks
DC24-  4A          LSR
DC25-  4A          LSR                 ; TOTAL BLOCKS HI / 16 = Anzahl der
DC26-  60          RTS                   existierenden VBM-Blocks -1
```

**– DEALLOCATE BLOCK**
(in der VBM)

Xref: $EBDD – DESTROY – den „höchstwertigen" Block erst nach Entfernen des Inhalts
$EC36 – DESTROY – Subdirectories Block per Block via Forward Reference
$ED84 – REMOVE BLOCKS – den vorhet „höchstwertigen" Block, danach Typ = Typ-1
$EDBA – REMOVE BLOCKS – den Inhalt eines Index/Masterblocks

```
DC27-  8E 5C F0    STX   $F05C         ; Blocknummer hi
DC2A-  48          PHA                 ; Blocknummer lo
DC2B-  AE 51 F0    LDX   $F051         ; VCB-Index
DC2E-  BD 13 F2    LDA   $F213,X       ; TOTAL BLOCKS hi
DC31-  CD 5C F0    CMP   $F05C         ; Test: wird hier versucht, einen Block
DC34-  68          PLA                   „out of Range" zu entfernen?
DC35-  90 6E       BCC   $DCA5         ; wenn ja, FILE STRUCTURE DAMAGED
DC37-  AA          TAX                 ; Blocknummer lo
DC38-  29 07       AND   #$07
DC3A-  A8          TAY
DC3B-  B9 C0 EF    LDA   $EFC0,Y       ; Bit Mask: $80/40/20/10/08/04/02/01
DC3E-  8D 5B F0    STA   $F05B         ; dieses Bit soll wieder gesetzt werden
DC41-  8A          TXA                 ; Reload Blocknummer lo
DC42-  4E 5C F0    LSR   $F05C
DC45-  6A          ROR
DC46-  4E 5C F0    LSR   $F05C
DC49-  6A          ROR
DC4A-  4E 5C F0    LSR   $F05C         ; Blocknummer/8 => Basis-Index innerhalb
DC4D-  6A          ROR                   eines VBM-Blocks
DC4E-  8D 62 F0    STA   $F062
DC51-  4E 5C F0    LSR   $F05C         ; Blockno hi/16 => VBM „extent" Nummer
DC54-  2E 64 F0    ROL   $F064         ; = Speicherseite im VBM-Block (0 oder 1)
DC57-  20 57 DD    JSR   $DD57         ; „rettet" die VBM eines anderen Units,
DC5A-  B0 48       BCS   $DCA4           falls noch im Speicher
DC5C-  AD 69 F0    LDA   $F069         ; „Extent Nummer" des mom. VBM-Blocks
DC5F-  CD 5C F0    CMP   $F05C         ; vs. „Extent", der benötigt wird
DC62-  F0 16       BEQ   $DC7A
DC64-  20 86 DD    JSR   $DD86         ; anderer Extent, u.U. vom selben Unit und
DC67-  B0 3B       BCS   $DCA4           nicht von $DD57 gerettet. Explizites WRITE
```

```
DC69-  AD 5C F0   LDA  $F05C       ; Nummer des benötigten Extents
DC6C-  AE 51 F0   LDX  $F051       ; VCB-Index
DC6F-  9D 1C F2   STA  $F21C,X     ; Setzen des momentanen Extents
DC72-  AD 66 F0   LDA  $F066       ; Unitnummer
DC75-  20 97 DD   JSR  $DD97       ; und Lesen des Extents (Blocknummer 00 06
DC78-  B0 2A      BCS  $DCA4         + Extent Nummer)
```

– benötigter VBM-Block (Extent) im Speicher

```
DC7A-  AC 62 F0   LDY  $F062       ; (Byte)-Basis des freizugebenden Blocks
DC7D-  4E 64 F0   LSR  $F064       ; Speicherseite (0 oder 1)
DC80-  AD 5B F0   LDA  $F05B       ; Mask mit dem zu setzenden Bit
DC83-  90 08      BCC  $DC8D       ; Carry set: Speicherseite 1
DC85-  19 00 F5   ORA  $F500,Y
DC88-  99 00 F5   STA  $F500,Y     ; Setzen des Bits (Freigeben des Blocks)
DC8B-  B0 06      BCS  $DC93
DC8D-  19 00 F4   ORA  $F400,Y
DC90-  99 00 F4   STA  $F400,Y
DC93-  A9 80      LDA  #$80
DC95-  0D 65 F0   ORA  $F065       ; Setzen des „VBM altered" Flag
DC98-  8D 65 F0   STA  $F065
DC9B-  EE 82 F0   INC  $F082       ; für REMOVE BLOCKS/DESTROY: Zähler der
DC9E-  D0 03      BNE  $DCA3         freigegebenen Blocks
DCA0-  EE 83 F0   INC  $F083
DCA3-  18         CLC
DCA4-  60         RTS

DCA5-  A9 5A      LDA  #$5A        ; ERR#5A: FILE STRUCTURE DAMAGED
DCA7-  38         SEC
DCA8-  60         RTS
```

**– ALLOCATE NEXT BLOCK**

– Suchen und Belegen des nächsten freien Blocks

Xref: $D536 – CREATE (Anhängen eines Blocks an eine Subdirectory)
$D66E – CREATE (Belegen eines Blocks für ein neues File/Subdirectory)
$E62E – WRITE (Weiterer Datenblock bzw. Master/Index bei Typerweiterung)

```
DCA9-  20 57 DD   JSR  $DD57       ; Lesen des (ersten) VBM-Blocks auf $F400
DCAC-  B0 23      BCS  $DCD1         und Rewrite anderer VBM-Blocks
DCAE-  A0 00      LDY  #$00
DCB0-  8C 64 F0   STY  $F064       ; Speicherseite 0 im VBM-Block
DCB3-  B9 00 F4   LDA  $F400,Y
DCB6-  D0 1A      BNE  $DCD2       ; freier Block gefunden
DCB8-  C8         INY
DCB9-  D0 F8      BNE  $DCB3       ; zuerst Absuchen der unteren Speicherseite
DCBB-  EE 64 F0   INC  $F064       ; => Speicherseite 1
DCBE-  EE 63 F0   INC  $F063       ; Blocknummer / $800 (VBM-Block/2)
DCC1-  B9 00 F5   LDA  $F500,Y
DCC4-  D0 0C      BNE  $DCD2       ; Absuchen der oberen Speicherseite
DCC6-  C8         INY
DCC7-  D0 F8      BNE  $DCC1
DCC9-  EE 63 F0   INC  $F063
DCCC-  20 35 DD   JSR  $DD35       ; wenn weitere VBM-Blocks möglich, Laden
DCCF-  90 DD      BCC  $DCAE         des nächsten VBM-Blocks und Loop
DCD1-  60         RTS              ; mit ERR#48: VOLUME FULL (von $DD35)

DCD2-  8C 62 F0   STY  $F062       ; freier Block gefunden: Index in der Seite
DCD5-  AD 63 F0   LDA  $F063       ; Blocknummer / $800
```

```
DCD8-  8D 47 F0     STA   $F047
DCDB-  98           TYA                 ; 1 VBM-Byte = 8 Blocks, d.h. der Index
DCDC-  0A           ASL                   innerhalb einer Seite * 8 ergibt die
DCDD-  2E 47 F0     ROL   $F047           Basis des freien Blocks
DCE0-  0A           ASL                   in $F047 (hi) und dem X-Reg (lo)
DCE1-  2E 47 F0     ROL   $F047
DCE4-  0A           ASL
DCE5-  2E 47 F0     ROL   $F047
DCE8-  AA           TAX                 ; X = $00/$08/$10/$18.....
DCE9-  38         * SEC                 ; Marker im 8-Block-Byte
DCEA-  AD 64 F0     LDA   $F064         ; Speicherseite: 0 oder 1
DCED-  F0 05        BEQ   $DCF4
DCEF-  B9 00 F5     LDA   $F500,Y
DCF2-  B0 03        BCS   $DCF7
DCF4-  B9 00 F4     LDA   $F400,Y
DCF7-  2A         * ROL                 ; den Marker hinein
DCF8-  B0 03        BCS   $DCFD
DCFA-  E8           INX                 ; die Blocknummer wird erhöht (0..7)
DCFB-  D0 FA        BNE   $DCF7         ; Loop, bis ein gesetztes Byte „abfällt"
DCFD-  4A           LSR                 ; Shift & Loop rückwärts, bis der Marker
DCFE-  90 FD        BCC   $DCFD           wieder erscheint
DD00-  8E 46 F0     STX   $F046         ; Blocknummer lo
DD03-  AE 64 F0     LDX   $F064         ; Speicherseite in der VBM
DD06-  D0 05        BNE   $DD0D
DD08-  99 00 F4     STA   $F400,Y       ; Zurück in die VBM. Das „abgefallene"
DD0B-  F0 03        BEQ   $DD10           Bit bleibt Null, d.h. der Block
DD0D-  99 00 F5     STA   $F500,Y         ist jetzt belegt.
DD10-  A9 80        LDA   #$80
DD12-  0D 65 F0     ORA   $F065         ; „VBM altered" Flag wird gesetzt
DD15-  8D 65 F0     STA   $F065
DD18-  AC 51 F0     LDY   $F051         ; VCB-Index für das Volume
DD1B-  B9 14 F2     LDA   $F214,Y
DD1E-  E9 01        SBC   #$01          ; die Anzahl der freien Blocks auf dem
DD20-  99 14 F2     STA   $F214,Y         Volume wird um eins erniedrigt
DD23-  B0 08        BCS   $DD2D
DD25-  B9 15 F2     LDA   $F215,Y
DD28-  E9 00        SBC   #$00
DD2A-  99 15 F2     STA   $F215,Y
DD2D-  18           CLC
DD2E-  AD 46 F0     LDA   $F046         ; Nummer des belegten Blocks lo
DD31-  AC 47 F0     LDY   $F047         ; hi
DD34-  60           RTS
```

– Test, ob mehrere VBM-Blocks auf dem Volume / ggf. Laden des nächsthöheren Extent

Xref: $DCCC – ALLOCATE NEXT BLOCK – kein freier Block im momentanen VBM-Block

```
DD35-  AC 51 F0     LDY   $F051         ; VCB-Index
DD38-  B9 13 F2     LDA   $F213,Y       ; TOTAL BLOCKS hi
DD3B-  4A           LSR
DD3C-  4A           LSR
DD3D-  4A           LSR
DD3E-  4A           LSR                 ; / 16 => Anzahl der VBM-Blocks
DD3F-  D9 1C F2     CMP   $F21C,Y       ; momentaner Extent
DD42-  F0 0F        BEQ   $DD53         ; bereits erreicht (letzter VBM-Block)?
DD44-  B9 1C F2     LDA   $F21C,Y
DD47-  18           CLC
DD48-  69 01        ADC   #$01          ; Setzen des nächsten Extent
DD4A-  99 1C F2     STA   $F21C,Y
DD4D-  20 86 DD     JSR   $DD86         ; Zurückschreiben des alten Extent
DD50-  4C 57 DD     JMP   $DD57         ; RTS via Lesen des neuen Extent
```

```
DD53-  A9 48     LDA  #$48        ; letzter VBM-Block bereits erreicht:
DD55-  38        SEC                 ERR#48: VOLUME FULL
DD56-  60        RTS
```

**– GET VBM**

– Lesen eines VBM-Blocks auf $F400/$F500

VBM-Blocks anderer Units werden zurückgeschrieben, wenn das Flag „VBM altered" gesetzt ist. RTS, wenn ein VBM-Block (beliebiger Extent!) des gewünschten Units bereits im Speicher ist.

Xref: $DC57 – DEALLOCATE BLOCK
$DCA9 – ALLOCATE NEXT BLOCK (Laden des ersten VBM-Blocks)
JP $DD50 – GET NEXT VBM BLOCK (s. oben)
$EBFC – DESTROY (Subdirectories)

```
DD57-  AC 51 F0  LDY  $F051       ; VCB-Index
DD5A-  B9 10 F2  LDA  $F210,Y     ; Unitnummer
DD5D-  CD 66 F0  CMP  $F066       ; Unitnummer des momentanen VBM-Blocks
DD60-  F0 0E     BEQ  $DD70
DD62-  20 86 DD  JSR  $DD86       ; ungleich: Rewrite des VBM-Blocks eines
DD65-  B0 1E     BCS  $DD85         anderen Units, wenn verändert
DD67-  AC 51 F0  LDY  $F051
DD6A-  B9 10 F2  LDA  $F210,Y     ; Setzen der neuen Unitnummer
DD6D-  8D 66 F0  STA  $F066       ; als „VBM-Block momentan im Speicher"

DD70-  AC 65 F0  LDY  $F065       ; „VBM altered" Flag, via $DD62 nie gesetzt
DD73-  30 05     BMI  $DD7A       ; VBM-Block verändert, nicht überschreiben!
DD75-  20 97 DD  JSR  $DD97       ; sonst Lesen des neuen VBM-Blocks
DD78-  B0 0B     BCS  $DD85
DD7A-  AC 51 F0  LDY  $F051
DD7D-  B9 1C F2  LDA  $F21C,Y     ; VBM Extent
DD80-  0A        ASL
DD81-  8D 63 F0  STA  $F063       ; VBM-Extent * 2 = Blocknummer / $800
DD84-  18        CLC
DD85-  60        RTS
```

**– REWRITE**

des alten VBM-Blocks, wenn verändert

Xref: $D68F/$DB84/$DC64/$DD4D/$DD62/$E081/$E09B

```
DD86-  18        CLC
DD87-  AD 65 F0  LDA  $F065       ; „VBM altered" Flag
DD8A-  10 F9     BPL  $DD85       ; nicht verändert: RTS
DD8C-  20 D9 DD  JSR  $DDD9       ; sonst Zurückschreiben auf das
DD8F-  B0 F4     BCS  $DD85         entsprechende Unit
DD91-  A9 00     LDA  #$00
DD93-  8D 65 F0  STA  $F065       ; und Reset des „VBM altered" Flags
DD96-  60        RTS
```

**– READ VBM**

(auf $F400/$F500)

Xref: $DC75 – DEALLOCATE BLOCK
$DD75 – GET VBM

```
DD97-   8D 66 F0    STA   $F066        ; Aufruf mit Unitnummer im Akku
DD9A-   AC 51 F0    LDY   $F051        ; VCB-Index
DD9D-   B9 1C F2    LDA   $F21C,Y      ; VBM-Extent Nummer (0,1,2,3...)
DDA0-   8D 69 F0    STA   $F069
DDA3-   18          CLC
DDA4-   79 1A F2    ADC   $F21A,Y      ; Nummer des ersten VBM-Blocks
DDA7-   8D 67 F0    STA   $F067
DDAA-   B9 1B F2    LDA   $F21B,Y      ; => $F067/68: Nummer des
DDAD-   69 00       ADC   #$00         ; momentanen VBM-Blocks
DDAF-   8D 68 F0    STA   $F068
DDB2-   A9 01       LDA   #$01         ; Befehl: READ
```

– Einsprung für WRITE von $DDDB

```
DDB4-   85 42       STA   $42          ; COMMAND für die RWTB
DDB6-   AD 30 BF    LDA   $BF30        ; Zwischenspeichern der Nummer
DDB9-   48          PHA                ;   des zuletzt benutzten Units
DDBA-   AD 66 F0    LDA   $F066        ; Unitnummer der VBM
DDBD-   8D 30 BF    STA   $BF30
DDC0-   AD 67 F0    LDA   $F067        ; Blocknummer für die RWTB
DDC3-   85 46       STA   $46
DDC5-   AD 68 F0    LDA   $F068
DDC8-   85 47       STA   $47
DDCA-   AD 8F DC    LDA   $DC8F        ; = $F4, d.h. Ziel/Quelladresse = $F400
DDCD-   20 E8 DD    JSR   $DDE8        ; Teil von R/W BLOCK von/nach $F400/$F600
DDD0-   AA          TAX                ; Eventuelle ERR#
DDD1-   68          PLA
DDD2-   8D 30 BF    STA   $BF30        ; Reload der „alten" Unitnummer
DDD5-   90 01       BCC   $DDD8
DDD7-   8A          TXA                ; ERR#
DDD8-   60          RTS
```

**– WRITE VBM-BLOCK**

Xref: $DD8C – Zurückschreiben eines geänderten VBM-Blocks

```
DDD9-   A9 02       LDA   #$02         ; Befehl: WRITE
DDDB-   D0 D7       BNE   $DDB4        ; „always"
```

**– WRITE BLOCK**
von $F600/$F700

Xref: $D549/$D568/$D5A1/$D67D/$D6F6/$D71B/$D759

```
DDDD-   A9 02       LDA   #$02         ; Befehl: WRITE
DDDF-   D0 02       BNE   $DDE3        ; "always„
```

**– READ BLOCK**
nach $F600/$F700

Xref: $D490/$D577/$D6D4/$D705/$D73F/$D773/$D81D
$E78B/$EB11/$EC0B/$EC45/$ECCC/$ED02/$ED36

```
DDE1-   A9 01       LDA   #$01         ; Befehl: READ

DDE3-   85 42       STA   $42          ; COMMAND für die RWTB
DDE5-   AD 2A D8    LDA   $D82A        ; = $F6
```

Xref: $DDCD – READ VBM (Zieladresse ist $F400/$F500!)

```
DDE8-  08          PHP
DDE9-  78          SEI
DDEA-  85 45       STA   $45          ; Ziel/Quelladresse hi
DDEC-  A9 00       LDA   #$00
DDEE-  85 44       STA   $44          ; Ziel/Quelladresse lo
DDF0-  8D 0F BF    STA   $BF0F        ; SYSERR auf 00
DDF3-  A9 FF       LDA   #$FF
DDF5-  8D 72 F0    STA   $F072        ; „Mystery Flag" - siehe READ, Stufe 3
DDF8-  AD 30 BF    LDA   $BF30
DDFB-  85 43       STA   $43          ; Unitnummer
DDFD-  20 DA D0    JSR   $D0DA        ; D0 DISK I/O
DE00-  B0 03       BCS   $DE05
DE02-  28          PLP                ; Reload des INT-Status
DE03-  18          CLC
DE04-  60          RTS
DE05-  28          PLP                ; Error Exit: Reload des INT-Status
DE06-  38          SEC
DE07-  60          RTS
```

**$CF – GET MARK COMMAND**; File muß OPEN sein – Set File Data vorher

```
DE08-  AE 52 F0    LDX   $F052        ; FCB-Index, gesetzt von Set File Data
DE0B-  A0 02       LDY   #$02
DE0D-  BD 12 F3    LDA   $F312,X      ; die momentane Position im File (3 Bytes)
DE10-  91 40       STA   ($40),Y        wird in den „user"-Puffer kopiert
DE12-  E8          INX
DE13-  C8          INY
DE14-  C0 05       CPY   #$05
DE16-  D0 F5       BNE   $DE0D
DE18-  18          CLC
DE19-  60          RTS
```

– Error Exit für SET MARK (nur für „user"-Command)

```
DE1A-  A9 4D       LDA   #$4D         ; ERR#4D: POSITION OUT OF RANGE
DE1C-  38          SEC
DE1D-  60          RTS
```

**$CE – SET MARK COMMAND**; File muß OPEN sein – Set File Data vorher

```
DE1E-  A0 04       LDY   #$04
DE20-  AE 52 F0    LDX   $F052        ; FCB-Index, gesetzt von Set File Data
DE23-  E8          INX
DE24-  E8          INX
DE25-  38          SEC                ; Marker für folgendes CMP
DE26-  B1 40       LDA   ($40),Y
DE28-  99 68 F0    STA   $F068,Y      ; die „Position gewünscht" wird aus dem
DE2B-  90 08       BCC   $DE35          „user"-Puffer kopiert und gleichzeitig
DE2D-  DD 15 F3    CMP   $F315,X      ; mit dem EOF verglichen
DE30-  90 03       BCC   $DE35        ; Sprung wenn „Position gewünscht" < EOF
DE32-  D0 E6       BNE   $DE1A        ; ERR#4D wenn „Position gewünscht" > EOF
DE34-  CA          DEX
DE35-  88          DEY
DE36-  98          TYA
DE37-  49 01       EOR   #$01         ; das Carry beibehalten!
DE39-  D0 EB       BNE   $DE26
```

**– POSITION (SET MARK)**

– Einsprung für alle internen Aufrufe

Xref: $E16E – OPEN: „Position jetzt" auf FF 00 00, „gewünscht" auf 00 00 00 => erzwingt das Lesen des ersten Blocks in den Filepuffer
$E24B – READ: Byte-per-Byte – nächster Block wird benötigt
$E275 – READ: Block-per-Block – nächster Block ins Zielgebiet (Pointer!)
JP $E303 – READ/WRITE gemeinsamer Exit:
READ: nach Stufe 2/3 Reload des Filepuffers
WRITE: wenn an Blockgrenze, wird der Block geschrieben
$E3B1 – READ: für Directories – nächster Block
$E4A6 – WRITE: nächster Block des Files - (Typ)Erweiterungsbits werden gesetzt, wenn File-Länge überschritten wird
$E939 – SET EOF: Erzwingt Reload des Filepuffers nach Verkürzung des Files, ebenfalls durch „Verdrehung" der „Positon jetzt"

```
DE3B-   AC 52 F0    LDY   $F052          ; FCB-Index
DE3E-   B9 13 F3    LDA   $F313,Y        ; „mittlere" Position jetzt = Blockno * 2
DE41-   29 FE       AND   #$FE
DE43-   8D 46 F0    STA   $F046
DE46-   AD 6B F0    LDA   $F06B          ; „Position gewünscht" – mittleres Byte
DE49-   38          SEC
DE4A-   ED 46 F0    SBC   $F046
DE4D-   8D 46 F0    STA   $F046          ; = Differenz in Blocks * 2
DE50-   90 0F       BCC   $DE61          ; negativ („rückwärts")
DE52-   C9 02       CMP   #$02
DE54-   B0 0B       BCS   $DE61          ; positiv > 2, d.h. > 1 Block
DE56-   AD 6C F0    LDA   $F06C
DE59-   D9 14 F3    CMP   $F314,Y        ; Hibytes 'jetzt/gewünscht'
DE5C-   D0 03       BNE   $DE61
DE5E-   4C 7C DF    JMP   $DF7C          ; ok. Block bereits im Speicher

DE61-   B9 07 F3    LDA   $F307,Y        ; Storage Type
DE64-   F0 07       BEQ   $DE6D          ; Sprung wenn 00 (deleted)
DE66-   C9 04       CMP   #$04
DE68-   90 0C       BCC   $DE76          ; ist Datenfile
DE6A-   4C AE DF    JMP   $DFAE          ; SET MARK für (Sub)diretories

DE6D-   A0 52       LDY   #$52       *** ; – Programmierfehler (s. „Spezielles")
DE6F-   99 00 F3    STA   $F300,Y
DE72-   A9 43       LDA   #$43           ; ERR#43: INVALID REFERENCE NUMBER
DE74-   38          SEC
DE75-   60          RTS
```

– Positionierung für die Typen 1/2/3

```
DE76-   B9 07 F3    LDA   $F307,Y        ; Storage Type
DE79-   8D 56 F0    STA   $F056
DE7C-   B9 08 F3    LDA   $F308,Y        ; STATUS im FCB-Eintrag
DE7F-   29 40       AND   #$40           ; geänderter Datenblock im Speicher?
DE81-   F0 05       BEQ   $DE88
DE83-   20 87 E0    JSR   $E087          ; ja, Zurückschreiben
DE86-   B0 69       BCS   $DEF1
DE88-   AC 52 F0    LDY   $F052          ; FCB-Index
DE8B-   B9 14 F3    LDA   $F314,Y        ; MSByte der „Position jetzt"
```

```
DE8E-   29 FE       AND   #$FE
DE90-   8D 46 F0    STA   $F046
DE93-   AD 6C F0    LDA   $F06C       ; MSByte „Position gewünscht"
DE96-   38          SEC               ; wenn die Differenz negativ oder > 2
DE97-   ED 46 F0    SBC   $F046         ist, wird ein anderer Indexblock benötigt
DE9A-   90 1E       BCC   $DEBA       ; negativ
DE9C-   C9 02       CMP   #$02
DE9E-   B0 1A       BCS   $DEBA       ; > 2
DEA0-   AE 56 F0    LDX   $F056       ; Positionierung im selben Indexblock:
DEA3-   CA          DEX               ; hat das File einen Storage Type >1?
DEA4-   D0 7A       BNE   $DF20       ; Ja, Sprung
```

– Laden des Datenblocks für Type 1 bzw. Typerweiterung

```
DEA6-   AD 6B F0    LDA   $F06B       ; „mittleres Byte" von „Position gewünscht"
DEA9-   4A          LSR
DEAA-   0D 6C F0    ORA   $F06C       ; MSByte „Position gewünscht"
DEAD-   D0 5F       BNE   $DF0E       ; Erweiterung, wenn „gewünscht" > 00 01 FF
DEAF-   B9 0C F3    LDA   $F30C,Y     ; „Position gewünscht" < 00 02 00: Bytes
DEB2-   85 46       STA   $46           $0C/$0D enthalten „Nummer erster Block"
DEB4-   B9 0D F3    LDA   $F30D,Y       d.h. die Nummer des Datenblocks
DEB7-   4C 72 DF    JMP   $DF72       ; Lesen => BLOCK IN MEMORY
```

– Differenz „jetzt" und „gewünscht" > 256 Blocks. Neuer Indexblock benötigt.
Ist die „Position gewünscht" innerhalb des zulässigen Bereichs, wird der Storage Type nicht heraufgesetzt. Wird von OPEN und SET EOF benutzt, um ein Lesen in den Filepuffer zu erzwingen.

```
DEBA-   B9 08 F3    LDA   $F308,Y     ; STATUS
DEBD-   29 80       AND   #$80        ; geänderter Indexblock im Speicher?
DEBF-   F0 05       BEQ   $DEC6
DEC1-   20 9B E0    JSR   $E09B       ; Ja, Zurückschreiben
DEC4-   B0 2B       BCS   $DEF1
DEC6-   AE 56 F0    LDX   $F056       ; Storage Type
DEC9-   E0 03       CPX   #$03
DECB-   F0 26       BEQ   $DEF3       ; Type 3, keine Erweiterung
DECD-   AD 6C F0    LDA   $F06C       ; MSByte „Position gewünscht"
DED0-   4A          LSR
DED1-   08          PHP               ; „Push Z-Flag"
DED2-   A9 07       LDA   #$07        ; Erweiterungsbits: 1 Master/1 Index/1 Data
DED4-   28          PLP
DED5-   D0 62       BNE   $DF39       ; MSByte „gewünscht" > 1 => Erweiterung
DED7-   20 A2 DF    JSR   $DFA2       ; im Bereich Typ 2: Erweiterungsbits löschen
DEDA-   CA          DEX
DEDB-   F0 C9       BEQ   $DEA6       ; File hat den Typ 1: Test auf Bereich
DEDD-   20 2E E0    JSR   $E02E       ; Type 2: Indexblock („erster Block") lesen
DEE0-   B0 0F       BCS   $DEF1
DEE2-   AC 52 F0    LDY   $F052       ; FCB-Index
DEE5-   A5 46       LDA   $46
DEE7-   99 0E F3    STA   $F30E,Y     ; Setzen der „Nummer Indexblock im Speicher"
DEEA-   A5 47       LDA   $47           im FCB-Eintrag
DEEC-   99 0F F3    STA   $F30F,Y
DEEF-   90 2F       BCC   $DF20       ; von $DEEC her ist das ein „always"
DEF1-   38          SEC
DEF2-   60          RTS
```

– Type 3: Lesen des Master-Blocks

```
DEF3-   20 A2 DF    JSR   $DFA2       ; Erweiterungsbits in STATUS löschen
DEF6-   20 2E E0    JSR   $E02E       ; Lesen des Masterblocks („erster Block")
DEF9-   B0 F6       BCS   $DEF1         in den Indexteil des Filepuffers
DEFB-   AD 6C F0    LDA   $F06C       ; MSByte „Position gewünscht"
```

```
DEFE-   4A          LSR             ; / 2 => Position des Indexblocks im Master
DEFF-   A8          TAY
DF00-   B1 48       LDA   ($48),Y   ; Hat dieser Indexblock die Nummer 00 00,
DF02-   E6 49       INC   $49         d.h. existiert er (noch) nicht, wird die
DF04-   D1 48       CMP   ($48),Y     Typerweiterung benutzt, um einen neuen
DF06-   D0 0B       BNE   $DF13       Indexblock zu erstellen
DF08-   C9 00       CMP   #$00
DF0A-   D0 07       BNE   $DF13     ; Indexblock existiert, Sprung
DF0C-   C6 49       DEC   $49
```

– neuer Indexblock benötigt (Type 3 und Erweiterung Type 1 => Type 2)

```
DF0E-   A9 03       LDA   #$03      ; Erweiterungsbits: 1 Index/ 1 Datenblock
DF10-   4C 39 DF    JMP   $DF39
```

– Type 3: Setzen der Indexblocknummer

```
DF13-   85 46       STA   $46       ; Setzen der Nummer des Indexblocks
DF15-   B1 48       LDA   ($48),Y
DF17-   85 47       STA   $47
DF19-   C6 49       DEC   $49
DF1B-   20 10 E0    JSR   $E010     ; Lesen des Indexblocks
DF1E-   B0 D1       BCS   $DEF1
```

– Indexblock im Speicher (Typen 2 und 3)

```
DF20-   AD 6C F0    LDA   $F06C     ; MSByte „Position gewünscht"
DF23-   4A          LSR
DF24-   AD 6B F0    LDA   $F06B     ; mittleres Byte
DF27-   6A          ROR             ; / 2 => Position des Datenblocks im Index
DF28-   A8          TAY
DF29-   B1 48       LDA   ($48),Y   ; derselbe Test wie beim Masterblock: hat
DF2B-   E6 49       INC   $49         der Datenblock die Nummer 00 00,
DF2D-   D1 48       CMP   ($48),Y     dann existiert er noch nicht und muß
DF2F-   D0 3B       BNE   $DF6C       via Erweiterung erstellt werden
DF31-   C9 00       CMP   #$00
DF33-   D0 37       BNE   $DF6C     ; ok. Datenblock existiert
DF35-   A9 01       LDA   #$01      ; Erweiterungsbit Bit 0: 1 Datenblock
DF37-   C6 49       DEC   $49
```

– Erweiterung bzw. Erstellung eines neuen Index/Datenblocks: Je nach gesetzten Erweiterungsbits wird der Datenteil bzw. der Indexteil des Filepuffers gelöscht und dient als neue (nächster) Block.

```
DF39-   AC 52 F0    LDY   $F052     ; FCB-Index
DF3C-   19 08 F3    ORA   $F308,Y   ; Einsprung mit Erweiterungsbits im Akku
DF3F-   99 08 F3    STA   $F308,Y   ; => Erweiterungsbits in STATUS
DF42-   4A          LSR
DF43-   4A          LSR
DF44-   20 5A DF    JSR   $DF5A     ; Löschen des Datenteils
DF47-   90 33       BCC   $DF7C     ; Bit 1 nicht gesetzt: kein neuer Indexblc
DF49-   91 48       STA   ($48),Y
DF4B-   C8          INY             ; Löschen des Indexteils im Filepuffer
DF4C-   D0 FB       BNE   $DF49
DF4E-   E6 49       INC   $49
DF50-   91 48       STA   ($48),Y
DF52-   C8          INY
DF53-   D0 FB       BNE   $DF50
DF55-   C6 49       DEC   $49
DF57-   4C 7C DF    JMP   $DF7C     ; BLOCK IN MEMORY (hier leerer Block)
```

– Löschung des Datenteils im Filepuffer

Xref: $DF44 – Datenblock existiert (noch) nicht
$E3CB – READ für Directories: EOF überschritten

```
DF5A-   A9 00       LDA   #$00
DF5C-   A8          TAY
DF5D-   91 4A       STA   ($4A),Y       ; $004A/4B zeigen auf den Datenteil des
DF5F-   C8          INY                   Filepuffers
DF60-   D0 FB       BNE   $DF5D
DF62-   E6 4B       INC   $4B
DF64-   91 4A       STA   ($4A),Y
DF66-   C8          INY
DF67-   D0 FB       BNE   $DF64
DF69-   C6 4B       DEC   $4B
DF6B-   60          RTS
```

– Holen der Datenblocknummer aus einem gelesenen Indexblock

```
DF6C-   85 46       STA   $46           ; Einsprung mit Blocknummer low im Akku
DF6E-   B1 48       LDA   ($48),Y       ; und gesetztem Index: Blocknummer hi
DF70-   C6 49       DEC   $49
```

– Lesen des Datenblocks, weiter mit BLOCK IN MEMORY

Xref: $DEB7 – Blocknummer für Type 1 gesetzt

```
DF72-   85 47       STA   $47           ; Einsprung mit Blocknummer hi im Akku
DF74-   20 F7 DF    JSR   $DFF7 \       ; Lesen des Blocks in den Datenteil
DF77-   B0 28       BCS   $DFA1           des Filepuffers
DF79-   20 A2 DF    JSR   $DFA2         ; Reset der Erweiterungsbits in STATUS
```

**– BLOCK IN MEMORY**

Xref: $DE5E – JP vom SET MARK Entry – Datenblock ist bereits im Speicher
$DF57 – JP von SET MARK – nach Löschen der entsprechenden Filepuffer-Teile
$E3C8 – READ für Directories – EOF überschritten, Setzen der Position im FCB

```
DF7C-   AC 52 F0    LDY   $F052         ; FCB-Index
DF7F-   C8          INY
DF80-   C8          INY
DF81-   A2 02       LDX   #$02
DF83-   B9 12 F3    LDA   $F312,Y       ; Kopieren der alten „Position jetzt"
DF86-   9D 4D F0    STA   $F04D,X         (nur für den Error Exit von WRITE)
DF89-   BD 6A F0    LDA   $F06A,X
DF8C-   99 12 F3    STA   $F312,Y       ; neue „Position jetzt" in den FCB-Eintrag
DF8F-   88          DEY
DF90-   CA          DEX
DF91-   10 F0       BPL   $DF83
DF93-   18          CLC
DF94-   A5 4A       LDA   $4A           ; lo-Byte des Datenpointers im Filepuffer
DF96-   85 4C       STA   $4C           ; = 00
DF98-   AD 6B F0    LDA   $F06B
DF9B-   29 01       AND   #$01          ; $004C/4D auf eine Seite innerhalb des
DF9D-   65 4B       ADC   $4B             Datenteils im Filepuffer
```

```
DF9F-  85 4D      STA   $4D
DFA1-  60         RTS
```

– Reset der Erweiterungsbits in STATUS – keine neuen Blocks/Erweiterung benötigt

Xref: $DED7 – SET MARK: „neuer" Indexblock für Type 2 – Erzwingung durch OPEN/SET EOF
$DEF3 – SET MARK: File hat bereits die Type 3
$DF79 – SET MARK: „Position gewünscht" ist im Bereich des momentanen Type
$E91D – SET EOF: File wurde verkürzt – Erzwingung eines Reads

```
DFA2-  AC 52 F0   LDY   $F052       ; FCB-Index
DFA5-  B9 08 F3   LDA   $F308,Y     ; STATUS
DFA8-  29 F8      AND   #$F8        ; Erweiterungsbits: Bit 0/Bit 1/Bit 2
DFAA-  99 08 F3   STA   $F308,Y
DFAD-  60         RTS
```

## – SET MARK IN DIRECTORY FILES

Xref: $DE6A – SET MARK Entry: File hat nicht den Type 0/1/2/3

```
DFAE-  C9 0D      CMP   #$0D        ; Subdirectory oder VolDIR (s. GET FILE –
DFB0-  F0 05      BEQ   $DFB7         „Fälschung" des Storage Type)
DFB2-  A9 4A      LDA   #$4A        ; ERR=4A: INCOMPATIBLE FILE FORMAT
DFB4-  20 09 BF   JSR   $BF09       ; SET SYSERR ; RTS z. Caller v. SET MARK
DFB7-  AD 46 F0   LDA   $F046       ; Differenz „jetzt-gewünscht" in Blocks
DFBA-  4A         LSR                 * 2 => Teilung durch 2 ergibt
DFBB-  8D 5A F0   STA   $F05A       ; den Herunterzähler
DFBE-  B9 13 F3   LDA   $F313,Y     ; mittleres Byte „Position jetzt"
DFC1-  CD 6B F0   CMP   $F06B       ; mittleres Byte „Position gewünscht"
DFC4-  90 0E      BCC   $DFD4       ; Sprung, wenn „jetzt" < „gewünscht"

DFC6-  A0 00      LDY   #$00        ; Positionierung „rückwärts": Index zur
DFC8-  20 E2 DF   JSR   $DFE2         Backward Reference
DFCB-  B0 24      BCS   $DFF1
DFCD-  EE 5A F0   INC   $F05A       ; und Heraufzählen der (negativen) Differenz
DFD0-  10 F4      BPL   $DFC6       ; => BLOCK IN MEMORY
DFD2-  30 A8      BMI   $DF7C

DFD4-  A0 02      LDY   #$02        ; Index zur Forward Reference
DFD6-  20 E2 DF   JSR   $DFE2       ; Lesen des nächsten Blocks
DFD9-  B0 16      BCS   $DFF1
DFDB-  CE 5A F0   DEC   $F05A       ; Herunterzählen der (positiven) Differenz
DFDE-  D0 F4      BNE   $DFD4
DFE0-  F0 9A      BEQ   $DF7C       ; => BLOCK IN MEMORY
```

– Lesen des nächsten DIR-Blocks via Forward/Backward Reference

Xref: $DFC8/$DFD6 – s.o.

```
DFE2-  B1 4A      LDA   ($4A),Y     ; $004A/4B: Pointer zum Datenteil des
DFE4-  85 46      STA   $46           Filepuffers, in dem bereits ein
DFE6-  C8         INY                 Block der Directory (via OPEN) steht
DFE7-  D1 4A      CMP   ($4A),Y
DFE9-  D0 08      BNE   $DFF3
DFEB-  C9 00      CMP   #$00
DFED-  D0 04      BNE   $DFF3       ; Reference existiert
DFEF-  A9 4C      LDA   #$4C        ; Ref= 00 00, d.h. Anfang/Ende überschritten
```

```
DFF1-   38          SEC                 ERR#4C: EOF ENCOUNTERED
DFF2-   60          RTS
DFF3-   B1 4A       LDA   ($4A),Y
DFF5-   85 47       STA   $47           ; Nummer des nächsten DIR-Blocks
```

– **READ BLOCK** in den Datenteil des Filepuffers

Xref: $DF74 – SET MARK: Blocknummer gesetzt, danach BLOCK IN MEMORY
$E18A – OPEN für Directory-Files – Read des Key Blocks
(SET MARK kann dafür nicht benutzt werden, s. DFAE)

```
DFF7-   A9 01       LDA   #$01          ; Befehl: READ
DFF9-   85 42       STA   $42
DFFB-   A2 4A       LDX   #$4A          ; => $004A/4B: Datenteil des Filepuffers
DFFD-   20 54 E0    JSR   $E054         ; setzt die Zieladresse und liest den Block


E000-   B0 0D       BCS   $E00F
E002-   AC 52 F0    LDY   $F052         ; FCB-Index
E005-   A5 46       LDA   $46
E007-   99 10 F3    STA   $F310,Y       ; Setzen von „Nummer Datenblock im Speicher"
E00A-   A5 47       LDA   $47             im FCB-Eintrag
E00C-   99 11 F3    STA   $F311,Y
E00F-   60          RTS
```

–**BLOCKREAD/WRITE** innerhalb des Filepuffers

Xref: $DF1B – SET MARK: Lesen des Indexblocks

```
E010-   A9 01       LDA   #$01          ; Befehl: READ
E012-   85 42       STA   $42
E014-   A2 48       LDX   #$48          ; => $48/$49: Indexteil des Filepuffers
E016-   20 54 E0    JSR   $E054         ; setzt die Zieladresse und liest den Block
E019-   B0 0E       BCS   $E029
E01B-   AC 52 F0    LDY   $F052         ; FCB-Index
E01E-   A5 46       LDA   $46
E020-   99 0E F3    STA   $F30E,Y       ; Setzen von „Nummer Indexblock im Speicher"
E023-   A5 47       LDA   $47             im FCB-Eintrag
E025-   99 0F F3    STA   $F30F,Y
E028-   18          CLC
E029-   60          RTS
```

Xref: $E084 – JP nach Rewrite eines VBM-Blocks. $E081 wird aufgerufen von:
$E5C3 – WRITE: Rewrite eines (neuen) Masterblocks
$E617 – WRITE: Typ-Erweiterung

```
E02A-   A9 02       LDA   #$02          ; Befehl: WRITE
E02C-   D0 02       BNE   $E030
```

– **READ INDEX/MASTER BLOCK** in den Indexteil des Filepuffers

Xref: $DEDD – SET MARK: File hat Type 2, Lesen des Indexblocks
$DEF – SET MARK: File hat Type 3, Lesen des Masterblocks
$E5E1 – WRITE: Typ-Erweiterung – wenn bereits Type 2, wird der Indexblock zuerst gelesen

```
E02E-  A9 01      LDA   #$01          ; Befehl: READ

E030-  48         PHA
E031-  A9 0C      LDA   #$0C
E033-  0D 52 F0   ORA   $F052         ; FCB-Index
E036-  A8         TAY                 ; => FCB-Bytes $0C/$0D => „erster Block"
E037-  68         PLA                 ; COMMAND
E038-  A2 48      LDX   #$48          ; => $48/$49: Indexteil des Filepuffers
```

Xref: $E091 – (Re)write des momentanen Datenblocks (Blockno von „$F30E")
$E0A8 – (Re)write des momentanen Indexblocks (Blockno von „$F310")

```
E03A-  85 42      STA   $42           ; COMMAND für die RWTB
E03C-  B9 00 F3   LDA   $F300,Y       ; Y = FCB-Index + $0C/$0E oder $10
E03F-  85 46      STA   $46           ; Blocknummer lo
E041-  D9 01 F3   CMP   $F301,Y
E044-  D0 09      BNE   $E04F
E046-  C9 00      CMP   #$00
E048-  D0 05      BNE   $E04F
E04A-  A9 0C      LDA   #$0C          ; Blocknummer = 00
E04C-  20 0C BF   JSR   $BF0C         ; => SYSTEM DEATH mit ERR=0C
E04F-  B9 01 F3   LDA   $F301,Y
E052-  85 47      STA   $47           ; Blocknummer hi
```

Xref: $DFFD – READ BLOCK in den Datenteil des Filepuffers; X=$4A
$E016 – READ BLOCK (Master/beliebigen Indexblock) in den Indexteil; X = $48

```
E054-  08         PHP                 ; INT-Status
E055-  78         SEI
E056-  B5 00      LDA   $00,X         ; X= $48(Index) oder $4A(Datenteil)
E058-  85 44      STA   $44
E05A-  B5 01      LDA   $01,X
E05C-  85 45      STA   $45           ; $0044/45: Pointer für die RWTB
E05E-  AC 52 F0   LDY   $F052         ; FCB-Index
E061-  B9 01 F3   LDA   $F301,Y       ; Unitnummer
E064-  8D 30 BF   STA   $BF30
E067-  A9 FF      LDA   #$FF
E069-  8D 72 F0   STA   $F072         ; Mystery Flag - s. READ, Stufe 3
E06C-  AD 30 BF   LDA   $BF30
E06F-  85 43      STA   $43           ; Unitnummer für die RWTB
E071-  A9 00      LDA   #$00
E073-  8D 0F BF   STA   $BF0F         ; SYSERR = 00
E076-  20 DA D0   JSR   $D0DA         ; D0 DISK I/O
E079-  B0 03      BCS   $E07E
E07B-  28         PLP
E07C-  18         CLC
E07D-  60         RTS
E07E-  28         PLP                 ; Error Return
E07F-  38         SEC
E080-  60         RTS
```

Xref: $E5C3/$E617 – s. $E02A

```
E081-  20 86 DD   JSR  $DD86        ; Rewrite VBM-Block, wenn geändert
E084-  4C 2A E0   JMP  $E02A
```

**– (RE)WRITE CURRENT DATA BLOCK**

Xref: $DE83 – SET MARK: Rewrite eines geänderten Datenblocks vor neuem Read
$E740 – FLUSH: Rewrite, falls noch geänderter Datenblock im Filepuffer

```
E087-  A2 4A      LDX  #$4A         ; => $4A/$4B: Datenteil des Filepuffers
E089-  A9 10      LDA  #$10
E08B-  0D 52 F0   ORA  $F052        ; + FCB-Index => $F310
E08E-  A8         TAY               ; zeigt auf „momentaner Datenblock"
E08F-  A9 02      LDA  #$02         ; Befehl: WRITE
E091-  20 3A E0   JSR  $E03A        ; setzt Pointer und schreibt
E094-  B0 22      BCS  $E0B8
E096-  A9 BF      LDA  #$BF         ; Löschen von „Datablock altered" in STATUS
E098-  4C AF E0   JMP  $E0AF
```

**– (RE)WRITE CURRENT INDEX BLOCK**

Xref: $DEC1 – SET MARK: ein anderer Indexblock wird benötigt (Type 3)
$E74C – FLUSH: Rewrite, falls noch geänderter Indexblock im Filepuffer

```
E09B-  20 86 DD   JSR  $DD86        ; Rewrite eines VBM-Blocks, falls nötig
E09E-  A2 48      LDX  #$48         ; => $48/$49: Indexteil des Filepuffers
E0A0-  A9 0E      LDA  #$0E
E0A2-  0D 52 F0   ORA  $F052        ; + FCB-Index => $F30E
E0A5-  A8         TAY               ; zeigt auf „momentaner Indexblock"
E0A6-  A9 02      LDA  #$02         ; Befehl: WRITE
E0A8-  20 3A E0   JSR  $E03A        ; setzt Pointer und schreibt
E0AB-  B0 0B      BCS  $E0B8
E0AD-  A9 7F      LDA  #$7F         ; Löschen von „Indexblock altered" in STATUS

E0AF-  AC 52 F0   LDY  $F052        ; FCB-Index
E0B2-  39 08 F3   AND  $F308,Y
E0B5-  99 08 F3   STA  $F308,Y      ; STATUS
E0B8-  60         RTS
```

**$C0 – OPEN COMMAND**; Copy & Verify Pathname ($D27F) vorher

```
E0B9-  20 98 D7   JSR  $D798        ; GET FILE
E0BC-  90 04      BCC  $E0C2
E0BE-  C9 40      CMP  #$40         ; INVALID PATHNAME, d.h. OPEN VolDIR?
E0C0-  D0 07      BNE  $E0C9        ; andere Fehler werden nicht abgefangen
E0C2-  20 A9 E1   JSR  $E1A9        ; Sucht FCB nach Files mit WRITE ACCESS
E0C5-  90 04      BCC  $E0CB          ab. RTS mit SEC, wenn File schon OPEN
E0C7-  A9 50      LDA  #$50         ; ERR#50: FILE (ALREADY) OPEN
E0C9-  38         SEC
E0CA-  60         RTS
E0CB-  AC 52 F0   LDY  $F052        ; FCB-Index von $E1A9 (freier Eintrag im FCB)
E0CE-  AD 53 F0   LDA  $F053        ; „freier Eintrag gefunden" von $E1A9
E0D1-  D0 04      BNE  $E0D7
E0D3-  A9 42      LDA  #$42         ; ERR#42: FCB TABLE FULL
E0D5-  38         SEC
```

```
E0D6-  60          RTS
E0D7-  A2 1F       LDX   #$1F
E0D9-  A9 00       LDA   #$00
E0DB-  99 00 F3    STA   $F300,Y      ; Der FCB-Eintrag wird gelöscht. Dabei
E0DE-  C8          INY                  werden auch „Position jetzt", „Blocks
E0DF-  CA          DEX                  momentan im Speicher" etc auf Null gesetzt
E0E0-  10 F9       BPL   $E0DB
E0E2-  A9 06       LDA   #$06
E0E4-  AA          TAX
E0E5-  0D 52 F0    ORA   $F052        ; FCB-Index
E0E8-  A8          TAY                  $F301     <= $F019:    Unitnummer
E0E9-  BD 18 F0    LDA   $F018,X        $F302/03 <= $F01A/1B: Key Blockno DIR
E0EC-  99 00 F3    STA   $F300,Y        $F304/05 <= $F01C/1D: Blockno DIR-Eintrag
E0EF-  88          DEY                  $F306     <= $F01E:    Eintragsnummer
E0F0-  CA          DEX
E0F1-  D0 F6       BNE   $E0E9
E0F3-  AD 1F F0    LDA   $F01F        ; Storage Type ($10/$20/$30/$D0)
E0F6-  4A          LSR                  + Namenslänge
E0F7-  4A          LSR                  GET FILE setzt $D0 für VolDIRs, s. dort
E0F8-  4A          LSR
E0F9-  4A          LSR
E0FA-  AA          TAX
E0FB-  99 07 F3    STA   $F307,Y      ; Storage Type im FCB ($01/$02/$03/$0D)
E0FE-  AD 3D F0    LDA   $F03D        ; ACCESS des Files
E101-  29 03       AND   #$03         ; nur R/W ACCESS werden kopiert
E103-  E0 0D       CPX   #$0D         ; Storage Type: ist File (Sub)directory?
E105-  D0 02       BNE   $E109
E107-  29 01       AND   #$01         ; Ja, nur READ erlaubt
E109-  99 09 F3    STA   $F309,Y      ; ACCESS im FCB-Eintrag
E10C-  29 02       AND   #$02         ; WRITE erlaubt?
E10E-  F0 05       BEQ   $E115        ; Ja, Sprung
E110-  AD 57 F0    LDA   $F057        ; „File gefunden" von $E1A9 („READ ONLY")
E113-  D0 B2       BNE   $E0C7        ; Ja, Abbruch - s.„Spezielles"
E115-  AD 3C F0    LDA   $F03C        ; MLI VERSION des Files
E118-  CD FF BF    CMP   $BFFF        ; muß kleiner/gleich der momentanen sein
E11B-  A9 00       LDA   #$00
E11D-  B0 04       BCS   $E123
E11F-  A9 4A       LDA   #$4A         ; ERR#4A: INCOMPATIBLE FILE FORMAT
E121-  38          SEC
E122-  60          RTS
E123-  E0 04       CPX   #$04         ; Storage Type
E125-  90 04       BCC   $E12B        ; Sprung für die Typen 1/2/3
E127-  E0 0D       CPX   #$0D         ; (Sub)directory?
E129-  D0 F4       BNE   $E11F        ; weder noch
E12B-  A2 06       LDX   #$06
E12D-  85 47       STA   $47
E12F-  AD 52 F0    LDA   $F052
E132-  1D C8 EF    ORA   $EFC8,X      ;      ($0C/$0D/$18/$19/$15/$16/$17)
E135-  A8          TAY
E136-  BD 30 F0    LDA   $F030,X        $F30C/0D <= $F030/31: „erster Block"
E139-  99 00 F3    STA   $F300,Y        $F318/19 <= $F032/33: BLOCKS USED
E13C-  CA          DEX                  $F315-17 <= $F034-36: EOF des Files
E13D-  10 EE       BPL   $E12D
E13F-  85 46       STA   $46
E141-  AC 52 F0    LDY   $F052        ; FCB-Index
E144-  AD 5A F0    LDA   $F05A        ; FCB-Eintragsnummer von $E1A9 (Index/$20)
E147-  99 00 F3    STA   $F300,Y      ; $F300: FCB-Eintragsnummer („belegt"-Flag)
E14A-  20 D7 ED    JSR   $EDD7        ; ALLOCATE FILEBUFFER
E14D-  B0 24       BCS   $E173        ; Sprung bei Kollision mit System Bit Map
E14F-  20 E0 D3    JSR   $D3E0        ; Setzt die Pufferadresse und den VCB-Index
E152-  AD 94 BF    LDA   $BF94
E155-  99 1B F3    STA   $F31B,Y      ; LEVEL für FLUSH und CLOSE
E158-  B9 07 F3    LDA   $F307,Y      ; Storage Type
E15B-  C9 04       CMP   #$04
```

```
E15D-  B0 2B      BCS   $E18A        ; Sprung für Subdirectories
E15F-  A9 FF      LDA   #$FF         ; Die „Position jetzt" wird auf FF 00 00,
E161-  99 14 F3   STA   $F314,Y        die „Position gewünscht" auf 00 00 00
E164-  A0 02      LDY   #$02           gesetzt. Der folgende Aufruf von SET MARK
E166-  A9 00      LDA   #$00           erzwingt damit das Einlesen des ersten
E168-  99 6A F0   STA   $F06A,Y        Datenblocks (+ Indexblock, wenn Type > 1)
E16B-  88         DEY                  in den Filepuffer und setzt die ent-
E16C-  10 FA      BPL   $E168          sprechenden Blocknummern im FCB-Eintrag
E16E-  20 3B DE   JSR   $DE3B        ; SET MARK
E171-  90 1C      BCC   $E18F        ; ok.

E173-  48         PHA                ; Fehler: Zwischenspeichern der ERR#
E174-  AC 52 F0   LDY   $F052        ; FCB-Index
E177-  B9 0B F3   LDA   $F30B,Y      ; Index der Filepuffer-Startadresse
E17A-  F0 06      BEQ   $E182        ; kein Puffer zugeordnet (von $E14D)
E17C-  20 34 EE   JSR   $EE34        ; DEALLOCATE FILEBUFFER
E17F-  AC 52 F0   LDY   $F052
E182-  A9 00      LDA   #$00         ; Ungültigmachen des FCB-Eintrags
E184-  99 00 F3   STA   $F300,Y
E187-  68         PLA                ; ERR#
E188-  38         SEC
E189-  60         RTS
```

– OPEN (Sub)directories: Lesen des Key Blocks

```
E18A-  20 F7 DF   JSR   $DFF7        ; Lesen des „ersten Blocks im File"
E18D-  B0 E4      BCS   $E173          in den Datenteil des Filepuffers
```

– erster Block des Files im Filepuffer

```
E18F-  AE 51 F0   LDX   $F051        ; VCB-Index
E192-  FE 1E F2   INC   $F21E,X      ; Erhöhung der Anzahl der OPEN Files
E195-  BD 11 F2   LDA   $F211,X      ; und Setzen des „OPEN File(s)"-Flag
E198-  09 80      ORA   #$80           für dieses Volume
E19A-  9D 11 F2   STA   $F211,X
E19D-  AC 52 F0   LDY   $F052        ; FCB-Index
E1A0-  B9 00 F3   LDA   $F300,Y      ; FCB-Reference Nummer (FCB-Index/$20)
E1A3-  A0 05      LDY   #$05
E1A5-  91 40      STA   ($40),Y      ; die Refno wird in den Parameterblock
E1A7-  18         CLC                  des „user"-Programms gespeichert
E1A8-  60         RTS
```

**– SCAN FCB**

Sucht den FCB nach einem bestimmten File und einem freien Eintrag ab.
Wird der File im FCB gefunden, wird ein Flag ($F057) gesetzt.
Hat der File WRITE ACCESS, RTS mit Carry gesetzt.
Der erste freie FCB-Eintrag wird festgehalten (Nummer + Flag).

Xref: $E0C2 – OPEN: Ist File bereits OPEN?
$EABA – RENAME: nicht möglich bei offenen Files
$EB55 – DESTROY: nicht möglich bei offenen Files

```
E1A9-  A9 00      LDA   #$00
E1AB-  8D 5A F0   STA   $F05A        ; Nummer des gefundenen (freien) Eintrags
E1AE-  8D 57 F0   STA   $F057        ; Reset des Flag: „File gefunden"
E1B1-  8D 53 F0   STA   $F053        ; Reset des Flag: „freier Eintrag gefunden"
E1B4-  A8         TAY
E1B5-  AE 53 F0   LDX   $F053
E1B8-  D0 03      BNE   $E1BD        ; bereits ein freier Eintrag gefunden
E1BA-  EE 5A F0   INC   $F05A        ; ansonsten Nummer vor Test erhöhen
```

```
E1BD-  B9 00 F3   LDA  $F300,Y
E1C0-  D0 0D      BNE  $E1CF        ; Eintrag nicht frei: Sprung zum Vergleich
E1C2-  8A         TXA
E1C3-  D0 28      BNE  $E1ED        ; bereits freier Eintrag gefunden: NE
E1C5-  8C 52 F0   STY  $F052        ; ansonsten wird der FCB-Index gesetzt
E1C8-  A9 FF      LDA  #$FF
E1CA-  8D 53 F0   STA  $F053        ; und das Flag „freier Eintrag gefunden"
E1CD-  D0 1E      BNE  $E1ED        ; „always"
E1CF-  98         TYA               ; Vergleich von:
E1D0-  09 06      ORA  #$06           $F019    - $F300:    Unitnummer
E1D2-  A8         TAY                 $F01A/1B - $F301/02: Blockno DIR Key
E1D3-  A2 06      LDX  #$06           $F01C/1D - $F303/04: Blockno Parent DIR
E1D5-  B9 00 F3   LDA  $F300,Y        $F01E    - $F305:    Eintragsnummer
E1D8-  DD 18 F0   CMP  $F018,X        $F01F    - $F307:    Storage Type
E1DB-  D0 10      BNE  $E1ED
E1DD-  88         DEY
E1DE-  CA         DEX
E1DF-  D0 F4      BNE  $E1D5
E1E1-  EE 57 F0   INC  $F057        ; Setzen von „File gefunden"
E1E4-  B9 09 F3   LDA  $F309,Y      ; ACCESS im FCB
E1E7-  29 02      AND  #$02
E1E9-  F0 02      BEQ  $E1ED        ; kein WRITE ACCESS: Exit ohne Carry
E1EB-  38         SEC               ; WRITE ACCESS: Abbruch mit Carry set
E1EC-  60         RTS
E1ED-  98         TYA               ; nächster FCB-Eintrag (NEXT)
E1EE-  29 E0      AND  #$E0
E1F0-  18         CLC
E1F1-  69 20      ADC  #$20
E1F3-  D0 BF      BNE  $E1B4
E1F5-  18         CLC               ; File nicht gefunden (nicht OPEN)
E1F6-  60         RTS
```

### $CA – READ COMMAND; File muß OPEN sein – Set File Data vorher

```
E1F7-  20 03 E4   JSR  $E403        ; Zieladresse vom „user" => $004E/4F
E1FA-  20 E8 E3   JSR  $E3E8        ; setzt Pufferlänge & Herunterzähler
E1FD-  48         PHA               ; und kehrt mit $F309 (ACCESS) wieder
E1FE-  20 15 E4   JSR  $E415        ; Kopie „Pos. jetzt"/Xfer Req/CMP EOF
E201-  68         PLA               ; ACCESS
E202-  29 01      AND  #$01         ; READ erlaubt?
E204-  D0 04      BNE  $E20A
E206-  A9 4E      LDA  #$4E         ; ERR#4E: FILE ACCESS ERROR
E208-  D0 24      BNE  $E22E
E20A-  90 25      BCC  $E231        ; Carry von $E415 (Vergleich mit EOF)
E20C-  AC 52 F0   LDY  $F052        ; FCB-Index: „Position jetzt" + Request
E20F-  B9 15 F3   LDA  $F315,Y        sind größer als EOF. Der Transferzähler
E212-  ED 6A F0   SBC  $F06A          wird entsprechend begrenzt.
E215-  8D 9A F0   STA  $F09A
E218-  8D 6D F0   STA  $F06D        ; Transfer-Zähler lo
E21B-  B9 16 F3   LDA  $F316,Y
E21E-  ED 6B F0   SBC  $F06B        ; „Position jetzt" - mittleres Byte
E221-  8D 9B F0   STA  $F09B
E224-  8D 6E F0   STA  $F06E        ; Transfer-Zähler hi
E227-  0D 9A F0   ORA  $F09A
E22A-  D0 10      BNE  $E23C        ; bei der Begrenzung sind noch Bytes übrig
E22C-  A9 4C      LDA  #$4C         ; Begrenzt auf 00 Bytes Transfer:
E22E-  4C E8 E2   JMP  $E2E8        ; ERR#4C: EOF ENCOUNTERED

E231-  AD 9A F0   LDA  $F09A        ; Start des Transfer-Loop:
E234-  0D 9B F0   ORA  $F09B          sind noch Bytes zu übertragen?
E237-  D0 03      BNE  $E23C
E239-  4C EF E2   JMP  $E2EF        ; Nein, Exit
```

```
E23C-  20 6C EE   JSR  $EE6C        ; Check des Zielgebiets mit der Bit Map
E23F-  B0 ED      BCS  $E22E        ; Error Handler
E241-  20 0E E4   JSR  $E40E        ; holt den Storage Type aus dem FCB-Eintrag
E244-  C9 04      CMP  #$04
E246-  90 03      BCC  $E24B
E248-  4C B1 E3   JMP  $E3B1        ; READ für Directories
```

– Byte-per-Byte Transfer Loop

```
E24B-  20 3B DE   JSR  $DE3B        ; GET MARK
E24E-  B0 DE      BCS  $E22E        ; Error Handler
E250-  20 06 E3   JSR  $E306        ; setzt NEWLINE (falls nötig) und Pointer
E253-  20 30 E3   JSR  $E330        ; kopiert vom Filepuffer ins Zielgebiet
E256-  70 E1      BVS  $E239        ; NEWLINE oder Transfer-Ende: EXIT
E258-  B0 F1      BCS  $E24B        ; NEWLINE aktiv: nächster Block
E25A-  AD 6E F0   LDA  $F06E        ; NEWLINE nicht aktiv: Test, ob noch
E25D-  4A         LSR               ;   mindestens ein Block ($200 Byte) übrig
E25E-  F0 EB      BEQ  $E24B        ; nein, Rest ebenfalls Byte-per-Byte
E260-  8D 6F F0   STA  $F06F        ; Anzahl der übrigen Blocks
E263-  20 FC E7   JSR  $E7FC        ; holt STATUS des Files aus dem FCB
E266-  29 40      AND  #$40         ; „Datablock altered" (WRITE vor READ)?
E268-  D0 E1      BNE  $E24B        ; ja, SET MARK schreibt den Block zurück
E26A-  8D 72 F0   STA  $F072        ; nein, „Mystery Flag" auf 00

E26D-  A5 4E      LDA  $4E          ; die Pointer zum Datenteil des Filepuffers
E26F-  85 4A      STA  $4A          ;   werden auf den Zielbereich von READ
E271-  A5 4F      LDA  $4F          ;   gesetzt. Sie müssen am Ende von READ
E273-  85 4B      STA  $4B          ;   wieder auf den Filepuffer gesetzt werden!
```

– Block-per-Block via SET MARK („Stufe 2")

```
E275-  20 3B DE   JSR  $DE3B        ; SET MARK liest den nächsten Block
E278-  B0 69      BCS  $E2E3        ;   direkt ins Zielgebiet von READ (Pointer!)
```

– Start des Loop für Stufe 3

```
E27A-  E6 4B      INC  $4B          ; Zielpointer + $200
E27C-  E6 4B      INC  $4B
E27E-  CE 6E F0   DEC  $F06E        ; Transfer-Zähler - $200
E281-  CE 6E F0   DEC  $F06E
E284-  EE 6B F0   INC  $F06B        ; „Position gewünscht" + $200
E287-  EE 6B F0   INC  $F06B
E28A-  D0 09      BNE  $E295
E28C-  EE 6C F0   INC  $F06C        ; MSByte „Position gewünscht" muß ggf.
E28F-  AD 6C F0   LDA  $F06C        ;   erhöht werden: sind wir am Ende eines
E292-  49 01      EOR  #$01         ;   Indexblocks? (MSByte = xxxx xx10)
E294-  4A         LSR               * ; wenn ja, ist das Carry jetzt gesetzt
E295-  CE 6F F0   DEC  $F06F        ; Zähler für die übrigen Blocks
E298-  D0 0D      BNE  $E2A7
E29A-  20 A3 E3   JSR  $E3A3        ; keine ganzen Blocks mehr übrig, die
E29D-  AD 6D F0   LDA  $F06D        ;   Pointer werden zurückgesetzt, danach
E2A0-  0D 6E F0   ORA  $F06E        ;   Test, ob noch einzelne Bytes übrig sind
E2A3-  F0 4A      BEQ  $E2EF        ; nein, Exit
E2A5-  D0 A4      BNE  $E24B        ; ja, Transfer Byte-per-Byte
```

– Stufe 3: Test auf „direct access"

```
E2A7-  B0 CC      BCS  $E275      * ; neuer Indexblock benötigt => SET MARK
E2A9-  AD 6C F0   LDA  $F06C        ; MSByte „Position gewünscht"
E2AC-  4A         LSR
E2AD-  AD 6B F0   LDA  $F06B        ; mittleres Byte
E2B0-  6A         ROR
```

```
E2B1-  A8          TAY                ; / 2 => Index zur Blocknummer
E2B2-  B1 48       LDA   ($48),Y      ; aus dem Indexblock
E2B4-  85 46       STA   $46
E2B6-  E6 49       INC   $49
E2B8-  D1 48       CMP   ($48),Y      ; Hibyte der Blocknummer
E2BA-  D0 09       BNE   $E2C5
E2BC-  C9 00       CMP   #$00      ***
E2BE-  D0 05       BNE   $E2C5        ; Block existiert
E2C0-  8D 72 F0    STA   $F072        ; „sparse" Block: „Mystery Flag" auf 00
E2C3-  F0 03       BEQ   $E2C8        ; „always", Blockno = 00 00
E2C5-  B1 48       LDA   ($48),Y      ; Blocknummer hi
E2C7-  18          CLC             *** ; Indikator
E2C8-  C6 49       DEC   $49
E2CA-  B0 A9       BCS   $E275     *** ; „sparse" Block: SET MARK löscht Zielgebiet
E2CC-  85 47       STA   $47          ; Setzen der Blocknummer hi
E2CE-  AD 72 F0    LDA   $F072        ; „Mystery": wurde seit einem „sparse Block"
E2D1-  F0 A2       BEQ   $E275          nichts vom Volume gelesen => SET MARK
E2D3-  18          CLC
E2D4-  08          PHP                ; Save des INT-Status
E2D5-  78          SEI
E2D6-  A5 4B       LDA   $4B
E2D8-  85 45       STA   $45          ; Zieladresse für die RWTB
E2DA-  20 DA D0    JSR   $D0DA        ; D0 DISK I/O: RWTB-Aufruf via Unitnummer
E2DD-  B0 03       BCS   $E2E2
E2DF-  28          PLP
E2E0-  90 98       BCC   $E27A        ; „always": Loop

E2E2-  28          PLP                ; Reload des INT-Status
E2E3-  48          PHA                ; ERR#
E2E4-  20 A3 E3    JSR   $E3A3        ; Zurücksetzen der Pointer auf Filepuffer
E2E7-  68          PLA                ; ERR#
```

– Error Handler

Xref: innerhalb von READ: mehrere Sprünge hierher
$E4D4 – WRITE Error Exit

```
E2E8-  48          PHA                ; ERR#
E2E9-  20 EF E2    JSR   $E2EF        ; Xfer Count => „user"-Parameter, SET MARK
E2EC-  68          PLA
E2ED-  38          SEC
E2EE-  60          RTS
```

– Setzt den „Transfer Count" und bringt den Filepuffer auf den neuen Stand
Nach READ-Stufen 2 und 3 ist die „Position jetzt" im FCB mindestens einen ganzen Block von der „Position gewünscht" entfernt stehengeblieben, d.h. der nächste Aufruf von SET MARK lädt den Filepuffer wieder mit den der neuen Position im File entsprechenden Daten.

Xref: $E239 – JP READ Exit, Transfer beendet
$E2E9 – READ: Error Handler
$E3BE – READ für Directories: JSR nach Transfer-Ende, danach Test auf EOF
JP $E49C – WRITE Exit: Request Count = 00 00
JP $E51C – WRITE Exit: Falls auf Block-Ende => SET MARK für Rewrite des Blocks

```
E2EF-  A0 06       LDY   #$06         ; Bytes $06/$07 im „user"-Parameterblock
E2F1-  38          SEC                  für den Transfer Count
```

```
E2F2-   AD 9A F0    LDA   $F09A         ; Transfer Request
E2F5-   ED 6D F0    SBC   $F06D         ; minus Herunterzähler = Anzahl der
E2F8-   91 40       STA   ($40),Y         transferierten Bytes
E2FA-   C8          INY
E2FB-   AD 9B F0    LDA   $F09B
E2FE-   ED 6E F0    SBC   $F06E         ; Hibytes
E301-   91 40       STA   ($40),Y
E303-   4C 3B DE    JMP   $DE3B         ; SET MARK-RTS von da aus ($D078/intern)
```

– Justiert den Quell/Zielpointer „rückwärts", damit für den Transfer-Loop derselbe Index benutzt werden kann. Kopie von NEWLINE, wenn spezifiziert.

Xref: $E250 – READ: vor der Kopierroutine Stufe 1 ($E330)
$E3B6 – READ für Directories – vor der Kopierroutine ($E330)
$E514 – WRITE: vor der WRITE-Kopierroutine ($E51F)

```
E306-   AC 52 F0    LDY   $F052         ; FCB-Index
E309-   38          SEC
E30A-   A5 4E       LDA   $4E           ; Quell/Zielpointer lo
E30C-   ED 6A F0    SBC   $F06A         ; „Position jetzt" lo
E30F-   85 4E       STA   $4E
E311-   B0 02       BCS   $E315
E313-   C6 4F       DEC   $4F           ; Quell/Zielpointer hi
E315-   B9 1F F3    LDA   $F31F,Y       ; NEWLINE AND MASK
E318-   18          CLC
E319-   F0 0A       BEQ   $E325         ; nicht spezifiziert
E31B-   38          SEC                 ; Indikator: NEWLINE aktiv
E31C-   8D 71 F0    STA   $F071
E31F-   B9 0A F3    LDA   $F30A,Y       ; NEWLINE CHAR
E322-   8D 70 F0    STA   $F070
E325-   AC 6A F0    LDY   $F06A         ; Start-Index für Kopierroutine
E328-   A5 4A       LDA   $4A           ; Kopie des Ziel/Quellpointers, wird
E32A-   85 4C       STA   $4C             beim Loop verändert
E32C-   AE 6D F0    LDX   $F06D         ; Transferzähler lo
E32F-   60          RTS                 ; mit SEC, wenn NEWLINE aktiv ist
```

**– READ MOVER LOOP**

Aufruf mit SEC, wenn NEWLINE aktiv. Exit mit V-Flag set, wenn Abbruch via NEWLINE oder Transfer Request heruntergezählt.

Xref: $E253 – READ Byte-per-Byte für Datenfiles
$E3B9 – READ für Directories

```
E330-   8A          TXA                 ; Aufruf mit Transferzähler lo im X-Reg
E331-   D0 08       BNE   $E33B
E333-   AD 6E F0    LDA   $F06E         ; Transferzähler hi
E336-   F0 52       BEQ   $E38A         ; auf 00 heruntergezählt: Exit
E338-   CE 6E F0    DEC   $F06E
E33B-   CA          DEX
E33C-   B1 4C       LDA   ($4C),Y       ; $004C/4D: Quell-Pointer (Filepuffer)
E33E-   91 4E       STA   ($4E),Y       ; $004E/4F: Pointer ins Zielgebiet
E340-   8A          TXA
E341-   F0 1B       BEQ   $E35E         ; wenn letztes Byte im Transferzähler (lo)
E343-   B0 2E       BCS   $E373         ; wenn NEWLINE aktiv, Test
E345-   CA          DEX
E346-   C8          INY
E347-   D0 F3       BNE   $E33C         ; sonst weiter, bis Y-Index = 00
```

```
E349-  A5 4D       LDA   $4D
E34B-  E6 4F       INC   $4F          ; Zieladresse + $100
E34D-  EE 6B F0    INC   $F06B        ; „Position jetzt" + $100
E350-  D0 03       BNE   $E355
E352-  EE 6C F0    INC   $F06C        ; MSByte „Position jetzt"
E355-  E6 4D       INC   $4D          ; Quell-Pointer
E357-  45 4B       EOR   $4B          ; $4D EOR $4B => 00, wenn erste Seite
E359-  F0 E1       BEQ   $E33C          innerhalb des Transfers => 2. Seite
E35B-  B8          CLV                ; Indikator: nicht durch NL/Zähler beendet
E35C-  50 2F       BVC   $E38D        ; „always" - Filepuffer-Ende

E35E-  AD 6E F0    LDA   $F06E        ; Herunterzähler low = 00. Test Zähler hi
E361-  F0 1A       BEQ   $E37D        ; wenn auch 00, Exit mit V-Flag set
E363-  C8          INY                ; noch Bytes übrig:
E364-  D0 06       BNE   $E36C        ; wenn nicht Seitenende, NE
E366-  A5 4D       LDA   $4D
E368-  45 4B       EOR   $4B          ; erste oder zweite Seite des Transfers?
E36A-  D0 03       BNE   $E36F        ; Sprung wenn zweite Seite
E36C-  CE 6E F0    DEC   $F06E        ; erste Seite: Transferzähler hi minus 1
E36F-  88          DEY                ; „INY" rückgängig machen
E370-  4C 43 E3    JMP   $E343        ; nächste Seite

E373-  B1 4C       LDA   ($4C),Y      ; gerade kopiertes Byte
E375-  2D 71 F0    AND   $F071        ; NEWLINE AND MASK
E378-  4D 70 F0    EOR   $F070        ; NEWLINE CHAR
E37B-  D0 C8       BNE   $E345        ; kein NEWLINE: NE
E37D-  C8          INY
E37E-  D0 0A       BNE   $E38A        ; $004E/4F auf nächstes Byte setzen
E380-  E6 4F       INC   $4F
E382-  EE 6B F0    INC   $F06B        ; und „Position gewünscht" anpassen, falls
E385-  D0 03       BNE   $E38A          eine Seitengrenze überschritten wurde
E387-  EE 6C F0    INC   $F06C

E38A-  2C A2 E3    BIT   $E3A2        ; = $60, SET V-FLAG (Transfer-Ende)

E38D-  8C 6A F0    STY   $F06A        ; „Position gewünscht"- LoByte
E390-  70 01       BVS   $E393        ; wenn Exit durch Filepuffer-Ende (V clear)
E392-  E8          INX                  ist X um eins zu klein
E393-  8E 6D F0    STX   $F06D        ; zwischenspeichern
E396-  08          PHP
E397-  18          CLC
E398-  98          TYA
E399-  65 4E       ADC   $4E
E39B-  85 4E       STA   $4E          ; $004E/4F auf Position nach Ende
E39D-  90 02       BCC   $E3A1          der Kopieraktion
E39F-  E6 4F       INC   $4F
E3A1-  28          PLP                ; mit V bei Abbruch via NEWLINE oder Zähler
E3A2-  60          RTS                ; mit C falls NEWLINE aktiv
```

– Restore der Filepuffer Pointer
Bei READ Stufe 2 und 3 zeigen die Filepuffer-Pointer auf das Zielgebiet selber und müssen nach Ende von READ wieder auf den Filepuffer gesetzt werden.

Xref: $E29A – READ: keine weiteren Block-per-Block Transfers
$E2E4 – READ: Error in Stufe 2 oder 3

```
E3A3-  A5 4A       LDA   $4A          ; momentane Ziel-Pointer => $004E/4F, falls
E3A5-  85 4E       STA   $4E            noch Byte-per-Byte Transfers zu machen
E3A7-  A5 4B       LDA   $4B            sind
E3A9-  85 4F       STA   $4F
E3AB-  AC 52 F0    LDY   $F052        ; FCB-Index
E3AE-  4C E0 D3    JMP   $D3E0        ; Teil von Set File Data, setzt Pointer neu
```

– **READ** für nicht-File Typen
Der File muß „automatisch" den Typ Subdirectory haben – sonst Error bei OPEN

Xref: JP $E248 – READ: Storage Type ist nicht im Bereich 0/1/2/3

```
E3B1-  20 3B DE   JSR  $DE3B       ; SET MARK - Block lesen
E3B4-  B0 2F      BCS  $E3E5       ; READ Error Handler
E3B6-  20 06 E3   JSR  $E306       ; Pointer justieren/NEWLINE kopieren
E3B9-  20 30 E3   JSR  $E330       ; READ MOVER LOOP
E3BC-  50 F3      BVC  $E3B1       ; Loop bis NEWLINE oder Count=00
E3BE-  20 EF E2   JSR  $E2EF       ; Return des Transfer Count + SET MARK
E3C1-  90 20      BCC  $E3E3       ; RTS
E3C3-  C9 4C      CMP  #$4C        ; Fehler: EOF ENCOUNTERED?
E3C5-  38         SEC
E3C6-  D0 1C      BNE  $E3E4       ; nein, anderer Fehler
```

– **READ** Directories, ab hier: s.„Spezielles"!

```
E3C8-  20 7C DF   JSR  $DF7C       ; die Directory wurde vollständig gelesen,
E3CB-  20 5A DF   JSR  $DF5A         d.h. bis zum EOF. „Position jetzt" wird
E3CE-  A0 00      LDY  #$00          auf den EOF gesetzt ($DF7C), der Datenteil
E3D0-  AE 52 F0   LDX  $F052         des Filepuffers wird gelöscht ($DF5A).
E3D3-  BD 10 F3   LDA  $F310,X     ; Bytes $10/$11 des FCB/Eintrags
E3D6-  91 4A      STA  ($4A),Y       „Block jetzt im Speicher" werden
E3D8-  A9 00      LDA  #$00           als Byte 00/01 im Filepuffer gesetzt.
E3DA-  9D 10 F3   STA  $F310,X     ; und im FCB auf 00 00
E3DD-  E8         INX
E3DE-  C8         INY
E3DF-  C0 02      CPY  #$02
E3E1-  D0 F0      BNE  $E3D3       ; Loop
E3E3-  18         CLC
E3E4-  60         RTS
E3E5-  4C E8 E2   JMP  $E2E8       ; Error Handler
```

– Setzt die Zielpuffer-Länge aus dem Request Count und kehrt mit ACCESS zurück

Xref: $E1FA – READ Entry
$E47C – WRITE Entry

```
E3E8-  A0 04      LDY  #$04        ; Bytes $04 und $05:
E3EA-  B1 40      LDA  ($40),Y     ; Request Count aus d. „user"-Parameterblock
E3EC-  8D 9A F0   STA  $F09A       ; Pufferlänge
E3EF-  8D 6D F0   STA  $F06D       ; Herunterzähler
E3F2-  C8         INY
E3F3-  B1 40      LDA  ($40),Y
E3F5-  8D 9B F0   STA  $F09B       ; Hibytes
E3F8-  8D 6E F0   STA  $F06E
E3FB-  AC 52 F0   LDY  $F052       ; FCB-Index
E3FE-  B9 09 F3   LDA  $F309,Y     ; ACCESS
E401-  18         CLC
E402-  60         RTS
```

– Kopiert die Startadresse des „user"-Puffers auf $004E/4F und kehrt mit Storage Type zurück

Xref: $D42A – ONLINE Entry: Puffer für Volume-Namen (Storage Type wird ignoriert)
$E1F7 – READ Entry: Start des Zielgebiets
$E49F – WRITE: Startadresse der Daten, die geschrieben werden sollen

```
E403-  A0 02     LDY   #$02          ; Bytes $02/$03 aus dem Parameterblock
E405-  B1 40     LDA   ($40),Y         des „user"-Programms: Quell/Zielstart
E407-  85 4E     STA   $4E
E409-  C8        INY
E40A-  B1 40     LDA   ($40),Y
E40C-  85 4F     STA   $4F
```

**– GET STORAGE TYPE**
(aus dem FCB-Eintrag)

Xref: $E241 – READ: Datenfile oder Directory?
$E594 – WRITE: Type-Check nach erster Typerweiterung
$E5DA – WRITE: Unterscheidung im Erweiterungsprozeß '1 -> 2' und '2 -> 3'
$E61C – WRITE: Erhöhung des Storage Type im Erweiterungsprozeß
$E807 – SET EOF: Unterscheidung Datenfile – Directory

```
E40E-  AC 52 F0  LDY   $F052         ; FCB-Index
E411-  B9 07 F3  LDA   $F307,Y       ; Byte $07 im FCB-Eintrag: Storage Type
E414-  60        RTS
```

– Kopiert „Position jetzt" auf $F06A/6B/6C und testet „Position + Request" gegen EOF

Xref: $E1FE – READ: Test, ob Begrenzung des Request auf den EOF notwendig
$E480 – WRITE: Test, ob der EOF des Files verändert (vergrößert) werden muß

```
E415-  A2 00     LDX   #$00
E417-  AC 52 F0  LDY   $F052         ; FCB-Index
E41A-  18        CLC
E41B-  B9 12 F3  LDA   $F312,Y       ; „Position jetzt" aus dem FCB-Eintrag
E41E-  9D 6A F0  STA   $F06A,X
E421-  9D 4D F0  STA   $F04D,X
E424-  7D 9A F0  ADC   $F09A,X       ; Request Count - $F09C muß 00 sein
E427-  9D 46 F0  STA   $F046,X
E42A-  8A        TXA
E42B-  49 02     EOR   #$02          ; „CMP #02" unter Beibehaltung des Carry
E42D-  F0 04     BEQ   $E433
E42F-  C8        INY
E430-  E8        INX
E431-  D0 E8     BNE   $E41B
```

Xref: $E45B – WRITE Error Exit: Setzen des neuen EOF

```
E433-  BD 46 F0  LDA   $F046,X
E436-  D9 15 F3  CMP   $F315,Y       ; Vergleich mit dem EOF aus dem FCB-Eintrag
E439-  90 06     BCC   $E441         ; RTS mit Carry gelöscht, wenn der EOF
E43B-  D0 04     BNE   $E441                immer noch größer ist
E43D-  88        DEY
```

```
E43E-  CA          DEX
E43F-  10 F2       BPL   $E433
E441-  60          RTS                    ; wenn EOF kleiner, RTS mit Carry set
```

– Teil des WRITE Error Exit. Update von „Position jetzt" und EOF im FCB

```
E442-  20 74 E4    JSR   $E474            ; X = FCB-Index + $02
E445-  BD 4A F0    LDA   $F04A,X
E448-  99 15 F3    STA   $F315,Y          ; Setzen des neuen EOF im FCB
E44B-  BD 4D F0    LDA   $F04D,X
E44E-  99 12 F3    STA   $F312,Y          ; Setzen von „Position jetzt" im FCB
E451-  9D 46 F0    STA   $F046,X
E454-  88          DEY
E455-  CA          DEX
E456-  10 ED       BPL   $E445
E458-  20 74 E4    JSR   $E474            ; X = FCB-Index + 2
E45B-  20 33 E4    JSR   $E433      ## ; Vergleich „Position jetzt" mit EOF
```

Xref: $E45B – WRITE Error Exit: wenn „Position jetzt" > EOF, => neuer EOF
$E483 – WRITE: Setzen des EOF auf „Position jetzt" + Request, wenn der alte EOF überschritten wird

```
E45E-  20 74 E4    JSR   $E474            ; X = FCB-Index + 02
E461-  B9 15 F3    LDA   $F315,Y          ; Kopie des EOF aus dem FCB
E464-  9D 4A F0    STA   $F04A,X
E467-  90 06       BCC   $E46F      ## ; vom Vergleich „$F04A.. – EOF"
E469-  BD 46 F0    LDA   $F046,X
E46C-  99 15 F3    STA   $F315,Y          ; EOF wurde überschritten und wird mit
E46F-  88          DEY                      „Position jetzt" bzw. „Position jetzt"
E470-  CA          DEX                      + Request Count neu gesetzt
E471-  10 EE       BPL   $E461
E473-  60          RTS
```

– setzt X-Reg auf FCB-Index + 2

Xref: $E442/$E458/$E45E – s.o.

```
E474-  A9 02       LDA   #$02
E476-  AA          TAX
E477-  0D 52 F0    ORA   $F052            ; FCB-Index
E47A-  A8          TAY
E47B-  60          RTS
```

**$CB – WRITE COMMAND**; File muß OPEN sein – Set File Data vorher

```
E47C-  20 E8 E3    JSR   $E3E8            ; Request Count => Pufferlänge ($F09A/9B)
E47F-  48          PHA                    ; ACCESS des Files nach RTS
E480-  20 15 E4    JSR   $E415            ; kopiert „Position jetzt" & EOF, Vergleich
E483-  20 5E E4    JSR   $E45E            ; wenn EOF überschritten: Neusetzen
E486-  68          PLA                    ; ACCESS
E487-  29 02       AND   #$02             ; WRITE erlaubt (Bit 1 gesetzt)?
E489-  D0 04       BNE   $E48F
E48B-  A9 4E       LDA   #$4E             ; ERR#4E: FILE ACCESS ERROR
E48D-  D0 40       BNE   $E4CF            ; „always"

E48F-  20 4E E6    JSR   $E64E            ; „activate" – Test auf WRITE PROTECTED
E492-  B0 3B       BCS   $E4CF              des Volumes (Laufwerk)
E494-  AD 9A F0    LDA   $F09A            ; Request Count
```

```
E497-  0D 9B F0    ORA   $F09B
E49A-  D0 03       BNE   $E49F
E49C-  4C EF E2    JMP   $E2EF        ; Request Count = 00 00, Exit ohne Fehler
E49F-  20 03 E4    JSR   $E403        ; kopiert die Startadresse der Daten und
E4A2-  C9 04       CMP   #$04         ; kehrt zurück mit dem Storage Type
E4A4-  B0 E5       BCS   $E48B        ; nicht 0/1/2/3: FILE ACCESS ERROR
```

– Transfer Loop

```
E4A6-  20 3B DE    JSR   $DE3B        ; SET MARK
E4A9-  B0 24       BCS   $E4CF
E4AB-  20 FC E7    JSR   $E7FC        ; GET STATUS
E4AE-  29 07       AND   #$07         ; Erweiterungsbits sind Bit 0-2
E4B0-  F0 62       BEQ   $E514        ; keine Erweiterung notwendig
E4B2-  A0 00       LDY   #$00
E4B4-  C8          INY                ; Zählung der gesetzten Erweiterungsbits
E4B5-  4A          LSR                  in STATUS (max. 3)
E4B6-  D0 FC       BNE   $E4B4
E4B8-  8C 54 F0    STY   $F054        ; = benötigte Blocks (BLOCKS NEEDED) lo
E4BB-  8D 55 F0    STA   $F055        ; = BLOCKS NEEDED (Hibyte, immer 00)
E4BE-  20 66 DB    JSR   $DB66        ; SPACE ON VOL? (Test – keine Belegung)
E4C1-  B0 0C       BCS   $E4CF        ; Sprung, wenn nicht mehr genug Platz
E4C3-  20 FC E7    JSR   $E7FC        ; GET STATUS
E4C6-  29 04       AND   #$04         ; Erweiterung auf Type 3?
E4C8-  F0 0D       BEQ   $E4D7        ; nein
E4CA-  20 8F E5    JSR   $E58F        ; erweitert das File auf Type 3
E4CD-  90 14       BCC   $E4E3
E4CF-  48          PHA                ; Error Exit: Save der ERR#
E4D0-  20 42 E4    JSR   $E442        ; Error Handler: setzt EOF und FCB-Eintrag
E4D3-  68          PLA
E4D4-  4C E8 E2    JMP   $E2E8        ; Error Handler von READ
```

– Test, ob neuer Indexblock benötigt (Type 3 und Erweiterung Type 1=>2)

```
E4D7-  20 FC E7    JSR   $E7FC        ; GET STATUS
E4DA-  29 02       AND   #$02         ; (irgendein) Indexblock benötigt?
E4DC-  F0 05       BEQ   $E4E3        ; nein
E4DE-  20 DA E5    JSR   $E5DA        ; erstellt einen Indexblock
E4E1-  B0 EC       BCS   $E4CF          und erhöht ggf. den Storage Type
```

– Erweiterungen des Storage Type erledigt – File hat jetzt mindestens Type 2

```
E4E3-  20 2E E6    JSR   $E62E        ; belegt den nächsten freien Block
E4E6-  B0 E7       BCS   $E4CF              (Blocknummer auf $F046/47)
E4E8-  20 FC E7    JSR   $E7FC        ; GET STATUS
E4EB-  09 80       ORA   #$80         ; Setzen von „Indexblock altered"
E4ED-  29 F8       AND   #$F8         ; und Löschen der Erweiterungsbits
E4EF-  99 08 F3    STA   $F308,Y
E4F2-  AD 6C F0    LDA   $F06C        ; „Position jetzt" – MSByte
E4F5-  4A          LSR
E4F6-  AD 6B F0    LDA   $F06B        ; „Position jetzt" – mittleres Byte
E4F9-  6A          ROR
E4FA-  A8          TAY                ; /2 = Block-Position im Indexblock
E4FB-  E6 49       INC   $49
E4FD-  AD 47 F0    LDA   $F047        ; von ALLOCATE NEXT BLOCK auf Nummer
E500-  AA          TAX                  des nächsten freien Blocks gesetzt:
E501-  91 48       STA   ($48),Y        die Blocknummer wird im Indexblock
E503-  C6 49       DEC   $49            eingetragen. Der Datenteil des Filepuffers
E505-  AD 46 F0    LDA   $F046          ist bereits von SET MARK gelöscht worden.
E508-  91 48       STA   ($48),Y
E50A-  AC 52 F0    LDY   $F052        ; FCB-Index
E50D-  99 10 F3    STA   $F310,Y      ; Nummer „Datenblock momentan im Speicher"
```

```
E510-  8A          TXA
E511-  99 11 F3    STA   $F311,Y       ; Hibyte
```

– der zu beschreibende Datenblock ist im Speicher

```
E514-  20 06 E3    JSR   $E306         ; Quell-Pointer justieren (s.READ)
E517-  20 1F E5    JSR   $E51F         ; WRITE MOVER LOOP - ignoriert NEWLINE
E51A-  50 8A       BVC   $E4A6         ; noch Bytes übrig => nächstes SET MARK
E51C-  4C EF E2    JMP   $E2EF         ; gemeinsames Ende von READ und WRITE
```

## – WRITE MOVER LOOP

Xref: $E517 – s.o.

```
E51F-  8A          TXA                 ; Aufruf mit Transferzähler low in X-Reg
E520-  D0 08       BNE   $E52A
E522-  AD 6E F0    LDA   $F06E         ; Transferzähler hi
E525-  F0 45       BEQ   $E56C         ; Exit
E527-  CE 6E F0    DEC   $F06E         ; Transferzähler minus $100
E52A-  CA          DEX
E52B-  B1 4E       LDA   ($4E),Y       ; Quelle - justiert auf Seitengrenze
E52D-  91 4C       STA   ($4C),Y       ; Datenteil des Filepuffers
E52F-  8A          TXA
E530-  F0 18       BEQ   $E54A
E532-  C8          INY
E533-  D0 F5       BNE   $E52A         ; Loop bis zur nächsten Seitengrenze

E535-  A5 4D       LDA   $4D
E537-  E6 4F       INC   $4F           ; Quelle + $100
E539-  EE 6B F0    INC   $F06B         ; nächste „Position gewünscht" + $100
E53C-  D0 03       BNE   $E541
E53E-  EE 6C F0    INC   $F06C         ; MSByte „Position gewünscht"
E541-  E6 4D       INC   $4D
E543-  45 4B       EOR   $4B           ; war das die erste Speicherseite?
E545-  F0 E3       BEQ   $E52A         ; ja, nächste Seite
E547-  B8          CLV                 ; nein, Indikator: noch Bytes übrig
E548-  50 25       BVC   $E56F         ; „always" - nächster Block (SET MARK)

E54A-  AD 6E F0    LDA   $F06E         ; Transferzähler lo ist 00. Wenn der
E54D-  F0 10       BEQ   $E55F           Zähler hi auch 00 ist => Exit
E54F-  C8          INY                 ; nein, Index wird erhöht
E550-  D0 06       BNE   $E558         ; nächste Seite
E552-  A5 4D       LDA   $4D           ; $100 oder $200 Bytes transferiert:
E554-  45 4B       EOR   $4B             wenn das die zweiten $100 Bytes waren,
E556-  D0 03       BNE   $E55B           ist der Filepuffer voll => Sprung
E558-  CE 6E F0    DEC   $F06E         ; ansonsten Transferzähler minus $100
E55B-  88          DEY
E55C-  4C 32 E5    JMP   $E532         ; Loop bis Filepuffer voll/Zähler-Ende

E55F-  C8          INY                 ; Sprung hierher, wenn Zähler = 00 00
E560-  D0 0A       BNE   $E56C         ; wenn das auf einer Seitengrenze passiert,
E562-  E6 4F       INC   $4F             wird der Quell-Pointer und die
E564-  EE 6B F0    INC   $F06B           nächste „Position gewünscht" für SET MARK
E567-  D0 03       BNE   $E56C           (WRITE Exit) ein letztes Mal erhöht
E569-  EE 6C F0    INC   $F06C
E56C-  2C A2 E3    BIT   $E3A2         ; $60 => SET V-FLAG (Transfer-Ende)
E56F-  8C 6A F0    STY   $F06A
E572-  8E 6D F0    STX   $F06D
E575-  08          PHP                 ; „Push V-Flag"
E576-  20 FC E7    JSR   $E7FC         ; GET STATUS
E579-  09 50       ORA   #$50          ; Bit 6: „Datablock altered" - Bit 4: Flag
```

```
E57B-   99 08 F3    STA   $F308,Y        für „ACTIVATE" und FLUSH (File verändert)
E57E-   18          CLC
E57F-   AD 6A F0    LDA   $F06A
E582-   65 4E       ADC   $4E            ; Justieren des Quell-Pointers auf
E584-   85 4E       STA   $4E              Transfer-Ende + 1
E586-   90 02       BCC   $E58A
E588-   E6 4F       INC   $4F
E58A-   20 50 EC    JSR   $EC50          ; setzt Byte $1C: „File altered"
E58D-   28          PLP                  ; „Pull V-Flag"
E58E-   60          RTS
```

– Erweiterung auf Storage Type 3

Xref: $E4CA – WRITE: Erweiterungsbit Bit 2 in STATUS durch SET MARK gesetzt

```
E58F-   20 E7 E5    JSR   $E5E7          ; Erhöhung des Storage Type um 1
E592-   B0 52       BCS   $E5E6
E594-   20 0E E4    JSR   $E40E          ; holt Storage Type aus dem FCB
E597-   C9 03       CMP   #$03           ; jetzt Type 3?
E599-   F0 05       BEQ   $E5A0
E59B-   20 E7 E5    JSR   $E5E7          ; nein, noch eine Erhöhung
E59E-   B0 46       BCS   $E5E6
```

– Erstellung eines neuen/nächsten Indexblocks (und/oder Erweiterung Type 2 => 3)

```
E5A0-   20 2E E6    JSR   $E62E          ; belegt den nächsten freien Block
E5A3-   B0 41       BCS   $E5E6            für den neuen Indexblock
E5A5-   AD 6C F0    LDA   $F06C          ; „Position gewünscht" - MSByte
E5A8-   4A          LSR
E5A9-   A8          TAY                  ; => Indexblock-Position im Masterblock
E5AA-   AD 46 F0    LDA   $F046          ; Nummer des nächsten freien Blocks ($E62E)
E5AD-   AA          TAX
E5AE-   91 48       STA   ($48),Y        ; die Blocknummer wird im Indexteil des
E5B0-   E6 49       INC   $49              des Filepuffers eingetragen. Hatte der
E5B2-   AD 47 F0    LDA   $F047            File vorher den Type 2, wurde dieser Teil
E5B5-   91 48       STA   ($48),Y          von SET MARK gelöscht und enthält jetzt
E5B7-   C6 49       DEC   $49              einen einzigen verzeichneten Indexblock.
E5B9-   AC 52 F0    LDY   $F052          ; FCB-Index
E5BC-   99 0F F3    STA   $F30F,Y        ; Nummer „Indexblock momentan im Speicher"
E5BF-   8A          TXA
E5C0-   99 0E F3    STA   $F30E,Y        ; Blocknummer hi
E5C3-   20 81 E0    JSR   $E081          ; Rewrite des VBM-Blocks und des Indexteils
E5C6-   B0 1E       BCS   $E5E6            des Filepuffers (MASTER-Block)
E5C8-   A0 00       LDY   #$00
E5CA-   98          TYA
E5CB-   91 48       STA   ($48),Y        ; Löschen des Indexteils im Filepuffer:
E5CD-   C8          INY                    hier kommt der neue Indexblock hinein
E5CE-   D0 FB       BNE   $E5CB
E5D0-   E6 49       INC   $49
E5D2-   91 48       STA   ($48),Y
E5D4-   C8          INY                    der Indexblock ist damit belegt und
E5D5-   D0 FB       BNE   $E5D2            momentan noch leer
E5D7-   C6 49       DEC   $49
E5D9-   60          RTS
```

– wenn Type 1, Erweiterung auf Type 2, ansonsten Lesen des „ersten Blocks im File", d.h. des MASTER- oder Indexblocks

Xref: $E4DE – WRITE: SET MARK hat in STATUS die Erweiterungsbits Bit 0/Bit 1 gesetzt.

```
E5DA-  20 0E E4   JSR  $E40E        ; holt die momentane Storage Type
E5DD-  C9 01      CMP  #$01
E5DF-  F0 06      BEQ  $E5E7        ; Sprung für Type 1
E5E1-  20 2E E0   JSR  $E02E        ; sonst Lesen des „ersten Blocks im File"
E5E4-  90 BA      BCC  $E5A0
E5E6-  60         RTS
```

– Erhöhung der Storage Type um 1 – Typerweiterung

Xref: $E5DF – Type 1 => Type 2
$E5E4 – Type 2 => Type 3 oder neuer Indexblock für Type 3
$E58F/$E59B – Erweiterung auf Type 3

```
E5E7-  20 2E E6   JSR  $E62E        ; belegt den nächsten freien Block
E5EA-  B0 41      BCS  $E62D          (Nummer auf $F046/47)
E5EC-  AC 52 F0   LDY  $F052        ; FCB-Index
E5EF-  B9 0C F3   LDA  $F30C,Y      ; und setzt die Nummer des gerade belegten
E5F2-  48         PHA                 Blocks als „erster Block im File" und als
E5F3-  AD 46 F0   LDA  $F046          Nummer „Indexblock momentan im Speicher".
E5F6-  AA         TAX                 Die vorherige Nummer „erster Block"
E5F7-  99 0C F3   STA  $F30C,Y        wird zwischengespeichert.
E5FA-  B9 0D F3   LDA  $F30D,Y
E5FD-  48         PHA
E5FE-  AD 47 F0   LDA  $F047
E601-  99 0D F3   STA  $F30D,Y      ; „erster Block im File" - Hibyte
E604-  99 0F F3   STA  $F30F,Y      ; „Indexblock momentan im Speicher" - Hibyte
E607-  8A         TXA
E608-  99 0E F3   STA  $F30E,Y
E60B-  A0 00      LDY  #$00
E60D-  E6 49      INC  $49          ; der Indexteil des Filepuffers wurde von
E60F-  68         PLA                 SET MARK gelöscht. Die vorherige Nummer
E610-  91 48      STA  ($48),Y        des „ersten Blocks im File" landet
E612-  C6 49      DEC  $49            als erster Eintrag im Indexteil.
E614-  68         PLA
E615-  91 48      STA  ($48),Y
E617-  20 81 E0   JSR  $E081        ; Rewrite des Indexteils im Filepuffer
E61A-  B0 11      BCS  $E62D
E61C-  20 0E E4   JSR  $E40E        ; holt den Storage Type des Files
E61F-  69 01      ADC  #$01
E621-  99 07 F3   STA  $F307,Y      ; STORAGE TYPE WIRD ERHÖHT
E624-  B9 08 F3   LDA  $F308,Y      ; STATUS
E627-  09 08      ORA  #$08         ; Setzen von Bit 3, wird nirgendwo
E629-  99 08 F3   STA  $F308,Y        wieder ausgewertet (s. „Spezielles")
E62C-  18         CLC
E62D-  60         RTS
```

– Belegung des nächsten freien Blocks für ein File

Xref: $E4E3 – WRITE: Erweiterung erledigt, Datenblock benötigt
$E5A0 – WRITE: Erweiterung: neuer Indexblock
$E5E7 – WRITE: Erhöhung des Storage Type um 1, Master oder Indexblock benötigt

```
E62E-  20 A9 DC   JSR  $DCA9        ; ALLOCATE NEXT BLOCK
E631-  B0 1A      BCS  $E64D          (Blocknummer auf $F046/47)
E633-  20 FC E7   JSR  $E7FC        ; GET STATUS
```

```
E636-  09 10      ORA   #$10          ; Setzen von Bit 4 - „File geändert" -
E638-  99 08 F3   STA   $F308,Y        wird von „ACTIVATE" und FLUSH benutzt
E63B-  B9 18 F3   LDA   $F318,Y
E63E-  18         CLC                  ; unnötig
E63F-  69 01      ADC   #$01           ; Erhöhung von BLOCKS USED um eins
E641-  99 18 F3   STA   $F318,Y
E644-  B9 19 F3   LDA   $F319,Y
E647-  69 00      ADC   #$00
E649-  99 19 F3   STA   $F319,Y
E64C-  18         CLC
E64D-  60         RTS
```

– **„ACTIVATE":** Wenn noch nicht geschehen, Test des Ziel-Volumes auf WRITE PROTECTED und „READY"

Xref: $E48F – WRITE
$E81C – SET EOF

```
E64E-  20 FC E7   JSR   $E7FC          ; GET STATUS
E651-  29 F0      AND   #$F0           ; wurde für diesen File schon einmal
E653-  D0 F7      BNE   $E64C            seit OPEN auf das Volume geschrieben?
E655-  B9 01 F3   LDA   $F301,Y        ; wenn nicht, wird die Unitnummer gesetzt
E658-  8D 30 BF   STA   $BF30
```

Xref: $D513 – CREATE: Test WPROT und „READY" vor Anhängen eines Subdir-Blocks
$EB82 – DESTROY: Test WPROT und „READY" vorher

```
E65B-  85 43      STA   $43           ; Einsprung/Aufruf mit Unitnummer im Akku
E65D-  A5 47      LDA   $47
E65F-  48         PHA
E660-  A5 46      LDA   $46           ; „alte Blocknummer" wird
E662-  48         PHA                   zwischengespeichert
E663-  A9 00      LDA   #$00
E665-  85 42      STA   $42           ; „00"-COMMAND
E667-  85 46      STA   $46           ; Blocknummer: 00 00, damit auf keinen Fall
E669-  85 47      STA   $47             die maximale Blockno überschritten wird
E66B-  20 DA D0   JSR   $D0DA         ; DO DISK I/O
E66E-  AA         TAX                 ; eventuelle ERR#
E66F-  68         PLA
E670-  85 46      STA   $46
E672-  68         PLA                 ; Restore der „alten" Blocknummer
E673-  85 47      STA   $47
E675-  8A         TXA                 ; eventuelle ERR#
E676-  60         RTS                 ; mit Carry clear/set von der RWTB
```

**$CC – CLOSE COMMAND**; Vorher nur internes SET TIME

```
E677-  A0 01      LDY   #$01          ; File-Reference Number wird aus
E679-  B1 40      LDA   ($40),Y       ; dem „user"-Parameterblock geholt
E67B-  D0 35      BNE   $E6B2         ; CLOSE SPECIFIC
E67D-  8D 7E F0   STA   $F07E         ; Reference Number = 00 „ALL":
                                      ; Reset des ERR#-Speicher
E680-  A9 00      LDA   #$00          ; unnötig
E682-  8D 52 F0   STA   $F052         ; Setzen des FCB-Index
E685-  A8         TAY
E686-  B9 1B F3   LDA   $F31B,Y       ; „LEVEL" im FCB-Eintrag (s. „Spezielles")
E689-  CD 94 BF   CMP   $BF94
```

```
E68C-  90 15      BCC   $E6A3       ; LEVEL auf der Global Page kleiner: NE
E68E-  B9 00 F3   LDA   $F300,Y     ; gültiger FCB-Eintrag?
E691-  F0 10      BEQ   $E6A3       ; nein, NE
E693-  20 14 E7   JSR   $E714       ; FLUSH SPECIFIC für diesen Eintrag
E696-  B0 4D      BCS   $E6E5
E698-  20 B7 E6   JSR   $E6B7       ; CLOSE SPECIFIC für diesen Eintrag
E69B-  A0 01      LDY   #$01
E69D-  B1 40      LDA   ($40),Y     ; ist Reference Number = 00 (ALL)?
E69F-  F0 02      BEQ   $E6A3       ; ja, NE
E6A1-  B0 42      BCS   $E6E5
E6A3-  AD 52 F0   LDA   $F052       ; FCB-Index + $20
E6A6-  18         CLC
E6A7-  69 20      ADC   #$20
E6A9-  90 D7      BCC   $E682       ; Loop bis zum Ende des FCB
E6AB-  AD 7E F0   LDA   $F07E       ; Zwischenspeicher für ERR#
E6AE-  D0 01      BNE   $E6B1       ; RTS mit Carry set (von $E6A9) bei Fehlern
E6B0-  18         CLC
E6B1-  60         RTS
```

**– CLOSE SPECIFIC**

– Einsprung für CLOSE eines bestimmten Files

```
E6B2-  20 1C E7   JSR   $E71C       ; FLUSH SPECIFIC – bestimmtes File
E6B5-  B0 2E      BCS   $E6E5
```

Xref: $E698 – CLOSE ALL Loop

```
E6B7-  AC 52 F0   LDY   $F052       ; FCB-Index
E6BA-  B9 0B F3   LDA   $F30B,Y     ; Puffer-Index von der System Global Page
E6BD-  20 34 EE   JSR   $EE34       ; DEALLOCATE BUFFER
E6C0-  B0 23      BCS   $E6E5
E6C2-  A9 00      LDA   #$00
E6C4-  AC 52 F0   LDY   $F052
E6C7-  99 00 F3   STA   $F300,Y     ; der FCB-Eintrag wird ungültig gesetzt
E6CA-  B9 01 F3   LDA   $F301,Y
E6CD-  8D 30 BF   STA   $BF30       ; Unitnummer
E6D0-  20 69 DA   JSR   $DA69       ; sucht den VCB-Eintrag für den File
E6D3-  AE 51 F0   LDX   $F051       ; FCB-Index, von $DA69 gesetzt
E6D6-  DE 1E F2   DEC   $F21E,X     ; Anzahl der OPEN Files auf dem Volume
E6D9-  D0 08      BNE   $E6E3
E6DB-  BD 11 F2   LDA   $F211,X     ; keine OPEN Files mehr auf dem Volume:
E6DE-  29 7F      AND   #$7F        ; Reset des Flags „OPEN File(s)"
E6E0-  9D 11 F2   STA   $F211,X
E6E3-  18         CLC
E6E4-  60         RTS               ; => $D078 oder CLOSE ALL Loop
```

– Error-„Verlängerung"

```
E6E5-  4C ED E7   JMP   $E7ED       ; zum Error Handler für CLOSE und FLUSH
```

**$CD – FLUSH COMMAND**; Vorher nur internes SET TIME

```
E6E8-  A0 01      LDY   #$01        ; Reference Number: Parameter Nummer 1
E6EA-  B1 40      LDA   ($40),Y     ; aus dem „user"-Parameterblock
E6EC-  D0 2E      BNE   $E71C       ; FLUSH SPECIFIC
E6EE-  8D 7E F0   STA   $F07E       ; 00 = ALL, => Zwischenspeicher für ERR#
E6F1-  A9 00      LDA   #$00        ; unnötig
E6F3-  8D 52 F0   STA   $F052       ; FCB-Index
E6F6-  A8         TAY
E6F7-  B9 00 F3   LDA   $F300,Y
```

```
E6FA-   F0 05       BEQ   $E701          ; ungültiger Eintrag: NE
E6FC-   20 14 E7    JSR   $E714          ; FLUSH SPECIFIC
E6FF-   B0 10       BCS   $E711
E701-   AD 52 F0    LDA   $F052          ; FCB-Index + $20 (NEXT)
E704-   18          CLC
E705-   69 20       ADC   #$20
E707-   90 EA       BCC   $E6F3
E709-   18          CLC
E70A-   AD 7E F0    LDA   $F07E          ; Zwischenspeicher für ERR#
E70D-   F0 01       BEQ   $E710          ; keine Fehler während des Loops
E70F-   38          SEC
E710-   60          RTS
```

– Error-„Verlängerung"

```
E711-   4C ED E7    JMP   $E7ED          ; zum Error Handler für CLOSE und FLUSH
```

**– FLUSH SPECIFIC**

Xref: $E693 – CLOSE ALL Loop
$E6FC – FLUSH ALL Loop

```
E714-   20 E0 D3    JSR   $D3E0          ; Teil von „Set File Data" – setzt Puffer,
E717-   90 0D       BCC   $E726          ; Pointer etc.
E719-   4C ED E7    JMP   $E7ED          ; Error Handler
```

– Einsprung für FLUSH eines bestimmten Files

Xref: $E6B2 – CLOSE eines bestimmten Files
$E863 – SET EOF: der neue EOF ist größer als der alte EOF

```
E71C-   A9 00       LDA   #$00
E71E-   8D 7E F0    STA   $F07E          ; Zwischenspeicher für ERR#
E721-   20 C5 D3    JSR   $D3C5          ; „Set File Data" mit Test der Refno
E724-   B0 EB       BCS   $E711
E726-   B9 09 F3    LDA   $F309,Y        ; ACCESS
E729-   29 02       AND   #$02           ; ist WRITE erlaubt?
E72B-   F0 DC       BEQ   $E709          ; nein, CLC/RTS
E72D-   B9 1C F3    LDA   $F31C,Y        ; „File altered"?
E730-   30 07       BMI   $E739          ; ja, FLUSH „always"
E732-   20 FC E7    JSR   $E7FC          ; GET STATUS: wenn Bit 4 („ACTIVATEd") und
E735-   29 70       AND   #$70             Bit 6 („Datablock altered") beide ge-
E737-   F0 D0       BEQ   $E709            löscht sind => kein WRITE => CLC/RTS
E739-   20 FC E7    JSR   $E7FC          ; GET STATUS
E73C-   29 40       AND   #$40           ; „Datablock altered"?
E73E-   F0 05       BEQ   $E745
E740-   20 87 E0    JSR   $E087          ; Ja, Rewrite des Datenblocks (Filepuffer)
E743-   B0 CC       BCS   $E711
E745-   20 FC E7    JSR   $E7FC          ; GET STATUS
E748-   29 80       AND   #$80           ; „Indexblock altered"?
E74A-   F0 05       BEQ   $E751
E74C-   20 9B E0    JSR   $E09B          ; Ja, Rewrite des Indexblocks (Filepuffer)
E74F-   B0 C0       BCS   $E711
```

– „Zusammenbau" des File Info für das Neuschreiben des File-Eintrags in ENTRY & UPDATE

```
E751-   A9 06       LDA   #$06
E753-   AA          TAX
E754-   0D 52 F0    ORA   $F052          ; FCB-Index
```

```
E757-  A8          TAY
E758-  B9 00 F3    LDA   $F300,Y      ; Schritt 1: Kopie aus dem FCB-Eintrag
E75B-  9D 18 F0    STA   $F018,X        => $F019:    Unitnummer
E75E-  88          DEY                  => $F01A/1B: Key Blockno der Directory
E75F-  CA          DEX                  => $F01C/1D: Nummer des DIR-Blocks
E760-  D0 F6       BNE   $E758          => $F01E:    Eintragsnummer
E762-  8D 30 BF    STA   $BF30        ; Unitnummer
E765-  AD 1A F0    LDA   $F01A        ; Schritt 2: Laden des Key Blocks der
E768-  85 46       STA   $46            Directory, in dem das File steht
E76A-  AD 1B F0    LDA   $F01B
E76D-  85 47       STA   $47          ; Blocknummer gesetzt,
E76F-  20 E1 DD    JSR   $DDE1        ; Lesen des Key Blocks auf $F600/$F700
E772-  B0 9D       BCS   $E711
E774-  20 A0 D8    JSR   $D8A0        ; => $F011: Eintragslänge; $F006..:Parent Info
E777-  AD 1C F0    LDA   $F01C
E77A-  AC 1D F0    LDY   $F01D
E77D-  CD 1A F0    CMP   $F01A        ; wenn der Eintrag nicht im Key Block
E780-  D0 05       BNE   $E787          selber steht, muß jetzt der entsprechende
E782-  CC 1B F0    CPY   $F01B          Block („Parent Block") gelesen werden
E785-  F0 07       BEQ   $E78E
E787-  85 46       STA   $46
E789-  84 47       STY   $47
E78B-  20 E1 DD    JSR   $DDE1        ; Lesen des „Parent Block"

E78E-  20 92 D6    JSR   $D692        ; Schritt 3: Bestimmung des Eintrags
E791-  20 9D D7    JSR   $D79D        ; und Herauskopieren der gesamten Daten
E794-  AC 52 F0    LDY   $F052          auf $F01F...
E797-  B9 18 F3    LDA   $F318,Y
E79A-  8D 32 F0    STA   $F032        ; Überschreiben von BLOCKS USED
E79D-  B9 19 F3    LDA   $F319,Y
E7A0-  8D 33 F0    STA   $F033
E7A3-  A2 00       LDX   #$00
E7A5-  B9 15 F3    LDA   $F315,Y      ; Überschreiben des EOF
E7A8-  9D 34 F0    STA   $F034,X
E7AB-  E8          INX
E7AC-  E0 03       CPX   #$03
E7AE-  F0 09       BEQ   $E7B9
E7B0-  B9 0C F3    LDA   $F30C,Y
E7B3-  9D 2F F0    STA   $F02F,X      ; Überschreiben der Nummer des
E7B6-  C8          INY                  „ersten Blocks im File"
E7B7-  D0 EC       BNE   $E7A5
E7B9-  B9 05 F3    LDA   $F305,Y      ; = „$F307,Y", d.h. Storage Type
E7BC-  0A          ASL
E7BD-  0A          ASL
E7BE-  0A          ASL
E7BF-  0A          ASL
E7C0-  8D 46 F0    STA   $F046        ; Scratch
E7C3-  AD 1F F0    LDA   $F01F        ; Storage Type & Namenslänge aus der DIR
E7C6-  29 0F       AND   #$0F         ; = Namenslänge
E7C8-  0D 46 F0    ORA   $F046        ; + (eventuell veränderter Storage Type)
E7CB-  8D 1F F0    STA   $F01F
E7CE-  20 AB D6    JSR   $D6AB        ; ENTRY & UPDATE
E7D1-  B0 1A       BCS   $E7ED
E7D3-  AC 52 F0    LDY   $F052        ; FCB-Index
E7D6-  B9 1C F3    LDA   $F31C,Y      ; Zurücksetzen von „File altered"
E7D9-  29 7F       AND   #$7F
E7DB-  99 1C F3    STA   $F31C,Y
E7DE-  AD 19 F0    LDA   $F019        ; Unitnummer des Files
E7E1-  CD 66 F0    CMP   $F066        ; Unitnummer des „momentanen" VBM-Blocks
E7E4-  D0 05       BNE   $E7EB
E7E6-  20 86 DD    JSR   $DD86        ; Rewrite des VBM-Blocks
E7E9-  B0 02       BCS   $E7ED
E7EB-  18          CLC
E7EC-  60          RTS
```

**– ERROR HANDLER für CLOSE & FLUSH:**
Abfangen von Fehlern für die ALL-Loops

Xref: $E6E5 – JP von der „Verlängerung" bei CLOSE
$E711 – JP von der „Verlängerung" bei FLUSH
$E719 – JP von FLUSH SPECIFIC

```
E7ED-   A0 01       LDY   #$01
E7EF-   48          PHA                   ; ERR#
E7F0-   B1 40       LDA   ($40),Y         ; Reference-Nummer = 0 (ALL-Loop)?
E7F2-   D0 06       BNE   $E7FA           ; nein, RTS mit Carry set
E7F4-   18          CLC
E7F5-   68          PLA
E7F6-   8D 7E F0    STA   $F07E           ; ja, ERR# zwischenspeichern
E7F9-   60          RTS
E7FA-   68          PLA
E7FB-   60          RTS
```

**– GET STATUS** des „momentanen" Files

Xref: $E263/$E4AB/$E4C3/$E4D7/$E4E8/$E576/$E633

```
E7FC-   AC 52 F0    LDY   $F052           ; FCB-Index
E7FF-   B9 08 F3    LDA   $F308,Y         ; STATUS im FCB-Eintrag
E802-   60          RTS
```

– Error Exit von SET EOF

```
E803-   A9 4E       LDA   #$4E            ; ERR#4E: FILE ACCESS ERROR
E805-   38          SEC                     (auch FILE LOCKED)
E806-   60          RTS
```

**$D0 – SET EOF COMMAND**; File muß OPEN sein – Set File Data vorher

```
E807-   20 0E E4    JSR   $E40E           ; holt Storage Type des Files
E80A-   C9 04       CMP   #$04            ; SET EOF geht nicht für (Sub)directories
E80C-   B0 F5       BCS   $E803
E80E-   0A          ASL
E80F-   0A          ASL
E810-   0A          ASL
E811-   0A          ASL
E812-   8D 81 F0    STA   $F081           ; Storage Type ins höhere Nibble
E815-   B9 09 F3    LDA   $F309,Y         ; ACCESS (Y-Index von $E40E)
E818-   29 02       AND   #$02
E81A-   F0 E7       BEQ   $E803           ; WRITE nicht erlaubt: ACCESS ERROR
E81C-   20 4E E6    JSR   $E64E           ; „ACTIVATE" - Test auf „READY" und WPROT
E81F-   B0 E2       BCS   $E803           ; nicht „ready" oder schreibgeschützt
E821-   AC 52 F0    LDY   $F052           ; FCB-Index
E824-   C8          INY
E825-   C8          INY
E826-   A2 02       LDX   #$02
E828-   B9 15 F3    LDA   $F315,Y         ; der EOF wird auf $F04A/4B/4C kopiert
E82B-   9D 4A F0    STA   $F04A,X
E82E-   88          DEY
E82F-   CA          DEX
E830-   10 F6       BPL   $E828
E832-   A0 04       LDY   #$04            ; Parameter 04/05/06 aus dem
```

```
E834-  A2 02     LDX  #$02          Block des „user"-Programms: EOF gewünscht
E836-  B1 40     LDA  ($40),Y
E838-  9D 6A F0  STA  $F06A,X     ; wird auf $F06A/6B/6C kopiert
E83B-  88        DEY
E83C-  CA        DEX
E83D-  10 F7     BPL  $E836
E83F-  A2 02     LDX  #$02
E841-  BD 4A F0  LDA  $F04A,X     ; Vergleich „EOF jetzt" ($F04A..)
E844-  DD 6A F0  CMP  $F06A,X     ; mit „EOF gewünscht" ($F06A..)
E847-  90 05     BCC  $E84E
E849-  D0 18     BNE  $E863       ; wenn der „EOF gewünscht" größer als der
E84B-  CA        DEX                „EOF jetzt" ist, wird der EOF schlicht
E84C-  10 F3     BPL  $E841         via $E84E neu gesetzt, RTS => $D078
```

– Setzen des neuen EOF im FCB-Eintrag

Xref: $E874 bzw. $E84C – neuer EOF ist größer
$E945 – SET EOF (Verkürzung des Files) erledigt

```
E84E-  A0 04     LDY  #$04
E850-  AE 52 F0  LDX  $F052       ; FCB-Index
E853-  E8        INX
E854-  E8        INX
E855-  B1 40     LDA  ($40),Y     ; der „EOF gewünscht" aus dem Parameter-
E857-  9D 15 F3  STA  $F315,X       block des „user"-Programms wird in
E85A-  CA        DEX                den FCB-Eintrag kopiert
E85B-  88        DEY
E85C-  C0 02     CPY  #$02
E85E-  B0 F5     BCS  $E855
E860-  4C 50 EC  JMP  $EC50       ; setzt „File altered" in STATUS
```

– „EOF GEWÜNSCHT" < „EOF JETZT" => Verkürzung des Files

```
E863-  20 1C E7  JSR  $E71C       ; FLUSH des Files $- falls noch geänderte
E866-  B0 9E     BCS  $E806         Daten/Indexblocks im Filepuffer
E868-  A6 4B     LDX  $4B         ; SET MARK (FLUSH Exit) hat d. Daten-Pointer
E86A-  E8        INX                auf die „Position jetzt" (Seitengrenze)
E86B-  E8        INX                gesetzt: Reset auf Anfang des Datenteils,
E86C-  86 49     STX  $49           d.h. 2 Seiten oberhalb des Indexteils
E86E-  A6 4A     LDX  $4A         ; lo-Bytes Indexteil-Pointer =>
E870-  86 48     STX  $48         ; Datenteil-Pointer
E872-  AC 52 F0  LDY  $F052       ; FCB-Index
E875-  C8        INY
E876-  C8        INY
E877-  A2 02     LDX  #$02
E879-  B9 12 F3  LDA  $F312,Y     ; Vergleich „EOF gewünscht" mit
E87C-  DD 6A F0  CMP  $F06A,X     ; „Position jetzt"
E87F-  90 17     BCC  $E898
E881-  D0 04     BNE  $E887
E883-  88        DEY
E884-  CA        DEX
E885-  10 F2     BPL  $E879
E887-  AC 52 F0  LDY  $F052       ; FCB-Index
E88A-  A2 00     LDX  #$00        ; wenn „Position jetzt" > „EOF gewünscht",
E88C-  BD 6A F0  LDA  $F06A,X       wird die „Position jetzt" im FCB-Eintrag
E88F-  99 12 F3  STA  $F312,Y       direkt auf den neuen EOF gesetzt
E892-  C8        INY
E893-  E8        INX
E894-  E0 03     CPX  #$03
E896-  D0 F4     BNE  $E88C
```

```
E898-  AD 6A F0   LDA   $F06A        ; „EOF gewünscht" - lobyte
E89B-  8D 86 F0   STA   $F086
E89E-  AD 6B F0   LDA   $F06B        ; mittleres Byte
E8A1-  8D 85 F0   STA   $F085
E8A4-  29 01      AND   #$01         ; ergibt $F086/87 = 00, wenn auf Blockgrenze
E8A6-  8D 87 F0   STA   $F087
E8A9-  AD 6C F0   LDA   $F06C
E8AC-  4A         LSR
E8AD-  8D 84 F0   STA   $F084        ; UNTERE GRENZE IM MASTER-BLOCK
E8B0-  6E 85 F0   ROR   $F085        ; UNTERE GRENZE IM INDEX-BLOCK
E8B3-  AD 86 F0   LDA   $F086
E8B6-  0D 87 F0   ORA   $F087
E8B9-  D0 1D      BNE   $E8D8        ; Sprung, wenn nicht auf Blockgrenze
E8BB-  AD 85 F0   LDA   $F085        ; auf Blockgrenze: Korrektur der
E8BE-  38         SEC                  Untergrenzen. Beispiel: EOF = $400 =>
E8BF-  E9 01      SBC   #$01           es bleiben nur Blocks 00 und 01 übrig,
E8C1-  8D 85 F0   STA   $F085          Block 02 wird dann nicht benötigt
E8C4-  A9 02      LDA   #$02
E8C6-  B0 0D      BCS   $E8D5
E8C8-  CE 84 F0   DEC   $F084
E8CB-  10 08      BPL   $E8D5
E8CD-  A9 00      LDA   #$00         ; führt diese Korrektur zu einem Ergebnis
E8CF-  8D 84 F0   STA   $F084          kleiner Null, werden die Grenzen auf
E8D2-  8D 85 F0   STA   $F085          00 00 00 gesetzt.
E8D5-  8D 87 F0   STA   $F087

E8D8-  AC 52 F0   LDY   $F052        ; FCB-Index
E8DB-  B9 0C F3   LDA   $F30C,Y      ; wenn SET EOF die File-Länge kürzt, wird
E8DE-  8D 7F F0   STA   $F07F          dabei eventuell der Storage Type und
E8E1-  B9 0D F3   LDA   $F30D,Y        die Nummer „erster Block" geändert.
E8E4-  8D 80 F0   STA   $F080        ; $F07F/$F080: „erster Block im File"
E8E7-  A9 00      LDA   #$00
E8E9-  8D 82 F0   STA   $F082        ; $F082/83: Zähler für entfernte Blocks
E8EC-  8D 83 F0   STA   $F083
E8EF-  20 62 EC   JSR   $EC62        ; REMOVE BLOCKS
E8F2-  08         PHP
E8F3-  48         PHA                ; eventuelle ERR#
E8F4-  38         SEC
E8F5-  AC 52 F0   LDY   $F052        ; FCB-Index
E8F8-  A2 00      LDX   #$00
E8FA-  BD 7F F0   LDA   $F07F,X      ; Update der Nummer „erster Block" im FCB
E8FD-  99 0C F3   STA   $F30C,Y
E900-  B9 18 F3   LDA   $F318,Y      ; BLOCKS USED
E903-  FD 82 F0   SBC   $F082,X      ; minus Anzahl der entfernten Blocks
E906-  99 18 F3   STA   $F318,Y
E909-  C8         INY
E90A-  E8         INX
E90B-  8A         TXA
E90C-  29 01      AND   #$01         ; Carry nicht verändern - 16-Bit Subtraktion
E90E-  D0 EA      BNE   $E8FA

E910-  AD 81 F0   LDA   $F081        ; Storage Type des Files
E913-  4A         LSR
E914-  4A         LSR
E915-  4A         LSR
E916-  4A         LSR                ; zurück ins untere Nibble (s.$E80E)
E917-  AC 52 F0   LDY   $F052
E91A-  99 07 F3   STA   $F307,Y      ; Update im FCB
E91D-  20 A2 DF   JSR   $DFA2        ; löscht die Erweiterungsbits in STATUS
E920-  20 DD EB   JSR   $EBDD        ; erhöht FREE BLOCKS auf dem Volume um
E923-  AC 52 F0   LDY   $F052                die Anzahl der entfernten Blocks
E926-  C8         INY
E927-  C8         INY
E928-  A2 02      LDX   #$02
```

```
E92A-  B9 12 F3   LDA  $F312,Y      ; Kopie der „Position jetzt" aus dem FCB
E92D-  9D 6A F0   STA  $F06A,X      ; als „Position gewünscht"
E930-  49 80      EOR  #$80
E932-  99 12 F3   STA  $F312,Y      ; „Verdrehen" und Zurückschreiben.
E935-  88         DEY                 Folge: der nächste Aufruf von SET MARK
E936-  CA         DEX                 erzwingt ein neues Einlesen in den
E937-  10 F1      BPL  $E92A          Filepuffer
E939-  20 3B DE   JSR  $DE3B        ; SET MARK
E93C-  90 07      BCC  $E945
E93E-  AA         TAX
E93F-  68         PLA               ; Fehler: das hier sind noch ERR# und P-Reg
E940-  28         PLP                 von REMOVE BLOCKS
E941-  8A         TXA               ; ERR# wird gegen die ERR# von SET MARK
E942-  38         SEC                 ausgewechselt. Passiert sowohl bei REMOVE
E943-  08         PHP                 BLOCKS als auch bei SET MARK ein Fehler,
E944-  48         PHA                 wird nur der SET MARK-Fehler returnt

E945-  20 4E E8   JSR  $E84E        ; normaler Exit: setzt den neuen EOF im FCB
E948-  68         PLA               ; ERR# von REMOVE BLOCKS
E949-  28         PLP
E94A-  60         RTS               ; => $D078
```

**$D1 – GET EOF COMMAND**; File muß OPEN sein – Set File Data vorher

```
E94B-  AE 52 F0   LDX  $F052        ; FCB-Index, von Set File Data gesetzt
E94E-  A0 02      LDY  #$02
E950-  BD 15 F3   LDA  $F315,X      ; Kopie des EOF in den Parameterblock des
E953-  91 40      STA  ($40),Y      ; „user"-Programms
E955-  E8         INX
E956-  C8         INY
E957-  C0 05      CPY  #$05
E959-  D0 F5      BNE  $E950
E95B-  18         CLC
E95C-  60         RTS               ; => $D078
```

**$C9 – NEWLINE COMMAND**; File muß OPEN sein – Set File Data vorher

```
E95D-  A0 02      LDY  #$02         ; Byte $02 aus dem „user"-Parameterblock:
E95F-  B1 40      LDA  ($40),Y      ; NEWLINE AND MASK
E961-  AE 52 F0   LDX  $F052        ; FCB-Index, von Set File Data gesetzt
E964-  9D 1F F3   STA  $F31F,X      ; FCB-Eintrag, Byte $1F: AND MASK
E967-  C8         INY
E968-  B1 40      LDA  ($40),Y      ; Byte $03:
E96A-  9D 0A F3   STA  $F30A,X      ; FCB-Eintrag, Byte $0A: NEWLINE CHAR
E96D-  18         CLC
E96E-  60         RTS               ; => $D078
```

**$C4 – GET FILE INFO COMMAND**; Vorher Copy & Verify Pathname ($D27F)

```
E96F-  20 98 D7   JSR  $D798        ; GET FILE – sucht & lädt File-Eintrag
E972-  90 42      BCC  $E9B6
E974-  C9 40      CMP  #$40         ; INVALID PATHNAME, d.h. Ende in VolDIR?
E976-  38         SEC
E977-  D0 5A      BNE  $E9D3        ; andere Fehler werden nicht abgefangen
```

– Volume Directory

```
E979-  A9 F0     LDA  #$F0        ; der von GET FILE „getürkte" Storage Type
E97B-  8D 1F F0  STA  $F01F         wird wieder auf „VolDIR" gesetzt
E97E-  A9 00     LDA  #$00
E980-  8D 54 F0  STA  $F054       ; BLOCKS NEEDED auf 00 00
E983-  8D 55 F0  STA  $F055
E986-  AE 51 F0  LDX  $F051       ; VCB-Index, gesetzt von GET FILE
E989-  20 71 DB  JSR  $DB71       ; SPACE ON VOLUME
E98C-  AE 51 F0  LDX  $F051
E98F-  BD 15 F2  LDA  $F215,X     ; von SPACE ON VOLUME im VCB gesetzte
E992-  8D 55 F0  STA  $F055         Anzahl der freien Blocks
E995-  BD 14 F2  LDA  $F214,X     ; Lobyte
E998-  8D 54 F0  STA  $F054
E99B-  BD 13 F2  LDA  $F213,X     ; TOTAL BLOCKS (Gesamtzahl) gehen in das
E99E-  8D 3F F0  STA  $F03F       ; AUX-TYPE - Field des FILE INFO
E9A1-  48        PHA
E9A2-  BD 12 F2  LDA  $F212,X
E9A5-  8D 3E F0  STA  $F03E       ; Lobyte
E9A8-  38        SEC
E9A9-  ED 54 F0  SBC  $F054       ; TOTAL BLOCKS minus BLOCKS FREE ergibt
E9AC-  8D 32 F0  STA  $F032       ; BLOCKS USED, Ergebnis in BLOCKS-USED
E9AF-  68        PLA                des File Info
E9B0-  ED 55 F0  SBC  $F055
E9B3-  8D 33 F0  STA  $F033
```

– Subdirectories/Datenfiles

```
E9B6-  AD 1F F0  LDA  $F01F       ; Storage Type
E9B9-  4A        LSR
E9BA-  4A        LSR
E9BB-  4A        LSR
E9BC-  4A        LSR
E9BD-  8D 1F F0  STA  $F01F       ; die Namenslänge wird ignoriert
E9C0-  A0 11     LDY  #$11
E9C2-  B9 CC EF  LDA  $EFCC,Y     ; $10/$1F/$20/$00/$13/$14/$21/$22/$23/$24
E9C5-  29 7F     AND  #$7F                ** **  **
E9C7-  AA        TAX
E9C8-  BD 1F F0  LDA  $F01F,X     ; Kopie des Parametersatzes in den
E9CB-  91 40     STA  ($40),Y     ; Block des „user"-Programms
E9CD-  88        DEY                die mit ** markierten Bytes haben
E9CE-  C0 03     CPY  #$03          zusätzlich noch Bit 7 gesetzt
E9D0-  B0 F0     BCS  $E9C2         (s. SET FILE INFO)
E9D2-  18        CLC
E9D3-  60        RTS              ; => $D078
```

**$C3 – SET FILE INFO COMMAND**; Copy & Verify Pathname ($D27F) vorher

```
E9D4-  20 98 D7  JSR  $D798       ; GET FILE
E9D7-  B0 25     BCS  $E9FE
E9D9-  AD 95 BF  LDA  $BF95       ; BACKUP BIT
E9DC-  49 20     EOR  #$20        ; wird für ENTRY & UPDATE gesetzt, wenn
E9DE-  2D 3D F0  AND  $F03D       ; ACCCESS des Files das BUBIT gesetzt
E9E1-  29 20     AND  #$20          hat und BACKUP BIT auf der Global Page
E9E3-  8D 74 F0  STA  $F074         nicht gesetzt ist (s. „Spezielles")
E9E6-  A0 0D     LDY  #$0D
E9E8-  BE CC EF  LDX  $EFCC,Y     ; Adress-Übersetzungstabelle. Hier werden
E9EB-  30 05     BMI  $E9F2         die drei Bytes des NULL-FIELD über-
E9ED-  B1 40     LDA  ($40),Y       sprungen, die anderen Parameter aus dem
E9EF-  9D 1F F0  STA  $F01F,X       „user"-Block werden in das File Info
E9F2-  88        DEY                kopiert (s. GET FILE INFO).
E9F3-  C0 03     CPY  #$03          Es erfolgt keine Prüfung, außer für
E9F5-  B0 F1     BCS  $E9E8         den Storage Type !!
E9F7-  29 1C     AND  #$1C        ; letztes Byte: Storage Type + Namenslänge
```

```
E9F9-  F0 04      BEQ   $E9FF        ; Sprung für Storage Type 0(!)1/2/3
E9FB-  A9 4E      LDA   #$4E         ; ERR#4E: UNSUPPORTED STORAGE TYPE
E9FD-  38         SEC
E9FE-  60         RTS
E9FF-  A0 0B      LDY   #$0B         ; Parameter $0B.. aus dem „user"-Parmblock:
EA01-  B1 40      LDA   ($40),Y      ; MODIFICATION DATE & TIME
EA03-  F0 03      BEQ   $EA08
EA05-  4C BB D6   JMP   $D6BB        ; ENTRY & UPDATE ohne Sys Time
EA08-  4C AB D6   JMP   $D6AB        ; ENTRY & UPDATE mit System Time
```

## $C2 – RENAME COMMAND; Copy & Verify Pathname ($D27F) vorher

```
EA0B-  20 AB D7   JSR   $D7AB        ; FIND FILE
EA0E-  90 3D      BCC   $EA4D
EA10-  C9 40      CMP   #$40         ; INVALID PATHNAME, d.h. Volume Directory?
EA12-  D0 18      BNE   $EA2C        ; andere Fehler werden nicht maskiert
```

### – RENAME für Volumes

```
EA14-  20 35 EB   JSR   $EB35        ; setzt Pointer auf 2. Namen, Copy & Verify
EA17-  B0 13      BCS   $EA2C        ; SEC/RTS (INVALID PATHNAME)
EA19-  AC 00 F1   LDY   $F100        ; Länge des ersten Namens im neuen Path
EA1C-  C8         INY
EA1D-  B9 00 F1   LDA   $F100,Y      ; die Länge des zweiten Namens im neuen
EA20-  D0 7F      BNE   $EAA1          Path muß Null sein, d.h. nur ein Name
EA22-  AE 51 F0   LDX   $F051        ; VCB-Index, gesetzt von FIND FILE
EA25-  BD 11 F2   LDA   $F211,X      ; OPEN File(s) auf dem Volume?
EA28-  10 04      BPL   $EA2E
EA2A-  A9 50      LDA   #$50         ; ERR#50: FILE IS OPEN
EA2C-  38         SEC
EA2D-  60         RTS
EA2E-  A0 00      LDY   #$00
EA30-  B9 00 F1   LDA   $F100,Y
EA33-  09 F0      ORA   #$F0
EA35-  20 26 EB   JSR   $EB26        ; kopiert den neuen Namen und schreibt
EA38-  B0 69      BCS   $EAA3          den geladenen VolDIR Key Block zurück
EA3A-  A0 00      LDY   #$00
EA3C-  AE 51 F0   LDX   $F051
EA3F-  B9 00 F1   LDA   $F100,Y      ; danach wird der neue Name in den
EA42-  F0 07      BEQ   $EA4B          FCB-Eintrag kopiert
EA44-  9D 00 F2   STA   $F200,X
EA47-  C8         INY
EA48-  E8         INX
EA49-  D0 F4      BNE   $EA3F
EA4B-  18         CLC
EA4C-  60         RTS                ; => $D078
```

### – RENAME für Subdirectories/Datenfiles

```
EA4D-  20 43 EB   JSR   $EB43        ; holt den Startindex des alten Pathname
EA50-  B9 00 F1   LDA   $F100,Y
EA53-  9D 00 F6   STA   $F600,X      ; der alte Pathname wird in den Directory
EA56-  30 04      BMI   $EA5C          Space kopiert
EA58-  C8         INY
EA59-  E8         INX
EA5A-  D0 F4      BNE   $EA50
EA5C-  20 35 EB   JSR   $EB35        ; Pointer auf neuen Pathname, Copy & Verify
EA5F-  B0 42      BCS   $EAA3
EA61-  20 43 EB   JSR   $EB43        ; holt jetzt Startindex des neuen Pathname
EA64-  B9 00 F1   LDA   $F100,Y
EA67-  DD 00 F6   CMP   $F600,X      ; Vergleich beider Paths, bis ein unter-
```

```
EA6A-  08          PHP                    schiedlicher Name gefunden wird.
EA6B-  29 F0       AND   #$F0           ; wenn sich $00 ergibt, haben wir ein
EA6D-  D0 06       BNE   $EA75            Längenbyte, d.h. den nächsten Namen
EA6F-  8C 79 F0    STY   $F079            im Path. Die Start-Indizes der jeweiligen
EA72-  8E 7A F0    STX   $F07A            Namen im Path werden gespeichert
EA75-  28          PLP                  ; vom Vergleich
EA76-  D0 09       BNE   $EA81
EA78-  E8          INX
EA79-  C8          INY
EA7A-  B9 00 F1    LDA   $F100,Y
EA7D-  D0 E8       BNE   $EA67
EA7F-  18          CLC                  ; beide Paths total gleich: RTS ohne Fehler
EA80-  60          RTS
```

– verschiedene Namen im Path gefunden

```
EA81-  AC 79 F0    LDY   $F079          ; Start-Index des verschiedenen Namens im
EA84-  B9 00 F1    LDA   $F100,Y          neuen Path: + Namenslänge => Index zum
EA87-  38          SEC                    Längenbyte des darauffolgenden Namens
EA88-  6D 79 F0    ADC   $F079
EA8B-  A8          TAY
EA8C-  B9 00 F1    LDA   $F100,Y        ; das Längenbyte des nächsten Namens muß 00
EA8F-  D0 10       BNE   $EAA1            sein, d.h. der Unterschied zwischen den
EA91-  AE 7A F0    LDX   $F07A            beiden Paths muß im letzten Namen
EA94-  BD 00 F6    LDA   $F600,X          auftreten!
EA97-  38          SEC
EA98-  6D 7A F0    ADC   $F07A          ; derselbe Test für den alten Pathnamen
EA9B-  AA          TAX
EA9C-  BD 00 F6    LDA   $F600,X
EA9F-  F0 04       BEQ   $EAA5
EAA1-  A9 40       LDA   #$40           ; ERR#40: INVALID PATHNAME
EAA3-  38          SEC
EAA4-  60          RTS

EAA5-  20 AB D7    JSR   $D7AB          ; FIND FILE mit dem neuen Pathnamen
EAA8-  B0 04       BCS   $EAAE
EAAA-  A9 47       LDA   #$47           ; ERR#47: DUPLICATE FILENAME
EAAC-  38          SEC                    (File mit diesem Namen existiert bereits)
EAAD-  60          RTS

EAAE-  C9 46       CMP   #$46           ; war der Fehler „FILE NOT FOUND"?
EAB0-  D0 F1       BNE   $EAA3          ; nein, andere Fehler werden nicht maskiert
EAB2-  20 7F D2    JSR   $D27F          ; Copy & Verify Pathname holt den alten Path
EAB5-  20 98 D7    JSR   $D798          ; GET FILE
EAB8-  B0 E9       BCS   $EAA3
EABA-  20 A9 E1    JSR   $E1A9          ; sucht den FCB nach OPEN Files ab
EABD-  A9 50       LDA   #$50           ; ERR#50: FILE IS OPEN
EABF-  B0 E2       BCS   $EAA3          ; $E1A9 setzt Carry, wenn File gefunden
EAC1-  AD 3D F0    LDA   $F03D          ; ACCESS
EAC4-  29 40       AND   #$40           ; RENAME erlaubt?
EAC6-  D0 04       BNE   $EACC
EAC8-  A9 4E       LDA   #$4E           ; ERR#4E: FILE ACCESS ERROR
EACA-  38          SEC
EACB-  60          RTS
EACC-  AD 1F F0    LDA   $F01F          ; Storage Type
EACF-  29 F0       AND   #$F0
EAD1-  C9 D0       CMP   #$D0
EAD3-  F0 08       BEQ   $EADD          ; ist Subdirectory
EAD5-  C9 40       CMP   #$40
EAD7-  90 04       BCC   $EADD          ; oder Type 0/1/2/3
EAD9-  A9 4A       LDA   #$4A           ; ERR#4A: INCOMPATIBLE FILE FORMAT
EADB-  D0 ED       BNE   $EACA          ; „always"
EADD-  20 35 EB    JSR   $EB35          ; setzt Pointer auf neuen Path, Copy & Verify
```

```
EAE0-  B0 C1      BCS   $EAA3        ; Unsinn, bereits verifiziert ($EA5C)
EAE2-  AC 79 F0   LDY   $F079        ; Start-Index des unterschiedlichen Namens
EAE5-  BE 00 F1   LDX   $F100,Y
EAE8-  98         TYA
EAE9-  79 00 F1   ADC   $F100,Y      ; Startindex für Kopier-Loop von hinten
EAEC-  A8         TAY
EAED-  B9 00 F1   LDA   $F100,Y      ; der neue Name wird in das File Info
EAF0-  9D 1F F0   STA   $F01F,X        kopiert
EAF3-  88         DEY
EAF4-  CA         DEX
EAF5-  D0 F6      BNE   $EAED
EAF7-  AD 1F F0   LDA   $F01F        ; Storage Type
EAFA-  29 F0      AND   #$F0
EAFC-  AA         TAX
EAFD-  19 00 F1   ORA   $F100,Y      ; + Länge des neuen Namens
EB00-  8D 1F F0   STA   $F01F
EB03-  E0 D0      CPX   #$D0
EB05-  D0 1C      BNE   $EB23        ; keine Subdirectory
EB07-  AD 30 F0   LDA   $F030        ; bei Subdirs muß auch der Name im Key
EB0A-  85 46      STA   $46            Block geändert werden: die Blocknummer
EB0C-  AD 31 F0   LDA   $F031          „erster Block" aus dem File Info
EB0F-  85 47      STA   $47            wird gesetzt
EB11-  20 E1 DD   JSR   $DDE1        ; und der Key Block gelesen
EB14-  B0 8D      BCS   $EAA3
EB16-  AC 79 F0   LDY   $F079
EB19-  B9 00 F1   LDA   $F100,Y
EB1C-  09 E0      ORA   #$E0         ; => Storage Type im Header
EB1E-  20 26 EB   JSR   $EB26        ; Kopie des Namens, Rewrite des Blocks
EB21-  B0 80      BCS   $EAA3

EB23-  4C BB D6   JMP   $D6BB        ; ENTRY & UPDATE ohne System Time
```

– Kopiert einen neuen Namen in den Directory Space und schreibt den Block zurück

Xref: $EA35 – RENAME Volumes: neuer Volume-Name im Key Block
$EB1E – RENAME Subdirectories: neuer Name im Key Block

```
EB26-  A2 00      LDX   #$00         ; Aufruf mit Storage Type & Namenslänge
EB28-  9D 04 F6   STA   $F604,X
EB2B-  E8         INX
EB2C-  C8         INY
EB2D-  B9 00 F1   LDA   $F100,Y
EB30-  D0 F6      BNE   $EB28        ; Loop bis Namensende
EB32-  4C DD DD   JMP   $DDDD        ; Rewrite des Blocks
```

– Setzt den Pointer zum neuen Namen und ruft Copy & Verify Pathname auf

Xref: $EA14 – RENAME Volumes
$EA5C – RENAME Subdirs/Files: Prüfung des neuen Namens + Kopie für Vergleich
$EADD – RENAME Subdirs/Files: Holen des neuen Namens für Kopie in File Info

```
EB35-  A0 03      LDY   #$03         ; Parameter 03/04 aus dem „user"-Block
EB37-  B1 40      LDA   ($40),Y      ; Startadresse des neuen Pathnamens
EB39-  C8         INY
EB3A-  85 48      STA   $48          ; Pointer zum Namen für Copy & Verify
EB3C-  B1 40      LDA   ($40),Y
EB3E-  85 49      STA   $49
EB40-  4C 8A D2   JMP   $D28A        ; Einsprung in Copy & Verify mit Pointer set
```

– Lädt den Startindex des momentanen Pathname

Xref: $EA4D – RENAME Subdirs/Files: Vor der Kopie des neuen Namens für Vergleich
$EA61 – RENAME Subdirs/Files: Reload des Startindex vor endgültiger Kopie

```
EB43-   A0 00       LDY   #$00
EB45-   2C 7C F0    BIT   $F07C         ; FULL NAME Flag
EB48-   30 03       BMI   $EB4D
EB4A-   AC 9A BF    LDY   $BF9A         ; Pathname nicht vollständig, Start mit
EB4D-   A2 00       LDX   #$00            System PREFIX
EB4F-   60          RTS
```

**$C1 – DESTROY COMMAND**; Vorher Copy & Verify Pathname ($D27F)

```
EB50-   20 98 D7    JSR   $D798         ; GET FILE
EB53-   B0 4A       BCS   $EB9F         ; nicht gefunden oder VolDIR: RTS mit ERR
EB55-   20 A9 E1    JSR   $E1A9         ; sucht den FCB ab: ist dieses File OPEN?
EB58-   AD 57 F0    LDA   $F057         ; „File gefunden"-Flag, gesetzt von $E1A9
EB5B-   F0 04       BEQ   $EB61
EB5D-   A9 50       LDA   #$50          ; ERR#50: FILE IS OPEN
EB5F-   38          SEC
EB60-   60          RTS

EB61-   A9 00       LDA   #$00          ; Setzen von BLOCKS NEEDED = 00 00
EB63-   8D 54 F0    STA   $F054
EB66-   8D 55 F0    STA   $F055
EB69-   20 66 DB    JSR   $DB66         ; SPACE ON VOL liefert BLOCKS FREE
EB6C-   90 05       BCC   $EB73
EB6E-   C9 48       CMP   #$48          ; ERR#48: VOLUME FULL wird maskiert
EB70-   38          SEC
EB71-   D0 2C       BNE   $EB9F         ; andere Fehler nicht

EB73-   AD 3D F0    LDA   $F03D         ; ACCESS aus dem File Info
EB76-   29 80       AND   #$80          ; DESTROY erlaubt?
EB78-   D0 05       BNE   $EB7F
EB7A-   A9 4E       LDA   #$4E          ; ERR#4E: FILE ACCESS ERROR
EB7C-   20 09 BF    JSR   $BF09         ; SET SYSERR , RTS zu $D078
EB7F-   AD 30 BF    LDA   $BF30         ; Unitnummer, gesetzt von GET FILE
EB82-   20 5B E6    JSR   $E65B         ; Test auf „READY" und WRITE PROTECTED
EB85-   B0 18       BCS   $EB9F
EB87-   AD 30 F0    LDA   $F030         ; Nummer des „ersten Blocks im File"
EB8A-   8D 7F F0    STA   $F07F           wird von DESTROY stufenweise verändert,
EB8D-   AD 31 F0    LDA   $F031
EB90-   8D 80 F0    STA   $F080
EB93-   AD 1F F0    LDA   $F01F         ; genauso wie der Storage Type.
EB96-   29 F0       AND   #$F0
EB98-   C9 40       CMP   #$40          ; Datenfile (Type 1/2/3)?
EB9A-   90 04       BCC   $EBA0
EB9C-   4C F8 EB    JMP   $EBF8         ; nein, muß Subdirectory sein
EB9F-   60          RTS
```

– DESTROY für Datenfiles

```
EBA0-   8D 81 F0    STA   $F081         ; Storage Type
EBA3-   A2 05       LDX   #$05
EBA5-   A9 00       LDA   #$00          ; $F082/83: Zähler f. entfernte Blocks
EBA7-   9D 81 F0    STA   $F081,X         $F084: Start mit ersten Indexblock
```

```
EBAA-   CA          DEX                 $F085: Start mit ersten Datenblock
EBAB-   D0 FA       BNE   $EBA7          $F086/87: Start innerhalb des Datenblocks
EBAD-   A9 02       LDA   #$02        ; ist hier $200: kein Löschen im Speicher,
EBAF-   8D 87 F0    STA   $F087         weil der Block vollständig entfernt wird.
EBB2-   20 62 EC    JSR   $EC62       ; REMOVE BLOCKS
EBB5-   B0 E8       BCS   $EB9F
```

Xref: $EBB5 – DESTROY für Datenfiles
JP $EC31 – DESTROY für Subdirectories

```
EBB7-   AE 80 F0    LDX   $F080       ; „erster Block im File" wird jetzt
EBBA-   AD 7F F0    LDA   $F07F         ebenfalls entfernt:
EBBD-   20 27 DC    JSR   $DC27       ; DEALLOCATE BLOCK
EBC0-   B0 DD       BCS   $EB9F
EBC2-   A9 00       LDA   #$00
EBC4-   8D 1F F0    STA   $F01F       ; Storage Type auf 00 (deleted)
EBC7-   CD 13 F0    CMP   $F013       ; Anzahl der aktiven Einträge in der DIR:
EBCA-   D0 03       BNE   $EBCF       ; wenn lobyte = 00,
EBCC-   CE 14 F0    DEC   $F014       ; DEC Hibyte
EBCF-   CE 13 F0    DEC   $F013
EBD2-   20 86 DD    JSR   $DD86       ; schreibt die VBM zurück
EBD5-   B0 C8       BCS   $EB9F
EBD7-   20 DD EB    JSR   $EBDD       ; erhöht BLOCKS FREE im VCB entsprechend
EBDA-   4C AB D6    JMP   $D6AB       ; ENTRY & UPDATE mit Storage Type 0
```

–BLOCKS FREE im VCB-Eintrag wird um die Anzahl der entfernten Blocks erhöht

Xref: $E920 – SET EOF nach Verkürzung eines Files
$EBD7 – DESTROY

```
EBDD-   AC 51 F0    LDY   $F051       ; VCB-Index
EBE0-   AD 82 F0    LDA   $F082       ; Anzahl der entfernten Blocks - lobyte
EBE3-   79 14 F2    ADC   $F214,Y     ; BLOCKS FREE lo
EBE6-   99 14 F2    STA   $F214,Y
EBE9-   AD 83 F0    LDA   $F083       ; Hibytes
EBEC-   79 15 F2    ADC   $F215,Y
EBEF-   99 15 F2    STA   $F215,Y
EBF2-   A9 00       LDA   #$00
EBF4-   99 1C F2    STA   $F21C,Y     ; VBM-Extent 00 (erster VBM-Block)
EBF7-   60          RTS
```

– DESTROY für Nicht-File Typen

```
EBF8-   C9 D0       CMP   #$D0        ; Storage Type: Subdirectory?
EBFA-   D0 4F       BNE   $EC4B       ; nein, INCOMPATIBLE FILE FORMAT
EBFC-   20 57 DD    JSR   $DD57       ; GET VBM
EBFF-   B0 49       BCS   $EC4A
EC01-   AD 30 F0    LDA   $F030       ; die Nummer des „ersten Blocks im File"
EC04-   85 46       STA   $46           ist hier die Nummer des Key Blocks
EC06-   AD 31 F0    LDA   $F031         der Subdirectory (aus dem File Info)
EC09-   85 47       STA   $47
EC0B-   20 E1 DD    JSR   $DDE1       ; liest den Key Block auf $F600
EC0E-   B0 3A       BCS   $EC4A
EC10-   AD 25 F6    LDA   $F625       ; Anzahl der aktiven Einträge
EC13-   D0 05       BNE   $EC1A         muß 00 00 sein, sonst Fehler
EC15-   AD 26 F6    LDA   $F626
EC18-   F0 05       BEQ   $EC1F
EC1A-   A9 4E       LDA   #$4E        ; ERR#4E: FILE ACCESS ERROR
EC1C-   20 09 BF    JSR   $BF09       ; SET SYSERR, RTS zu $D078
EC1F-   8D 04 F6    STA   $F604       ; Storage Type/Namenslänge = 00
```

```
EC22-  20 DD DD    JSR   $DDDD          ; Rewrite des Blocks
EC25-  B0 23       BCS   $EC4A

EC27-  AD 02 F6    LDA   $F602          ; Forward Reference zum nächsten DIR-Block
EC2A-  CD 03 F6    CMP   $F603
EC2D-  D0 04       BNE   $EC33          ; wenn 00 00, ist die Directory zuende:
EC2F-  C9 00       CMP   #$00             der „erste Block" (Key Block) wird bei
EC31-  F0 84       BEQ   $EBB7            $EBB7 entfernt

EC33-  AE 03 F6    LDX   $F603
EC36-  20 27 DC    JSR   $DC27          ; DEALLOCATE BLOCK
EC39-  B0 0F       BCS   $EC4A
EC3B-  AD 02 F6    LDA   $F602          ; die Nummer des nächsten Blocks wird
EC3E-  85 46       STA   $46              gesetzt. Dieser Block ist soeben entfernt
EC40-  AD 03 F6    LDA   $F603            worden, d.h. der „erste Block im File"
EC43-  85 47       STA   $47              ist nach wie vor in der VBM belegt!
EC45-  20 E1 DD    JSR   $DDE1          ; liest den Block auf $F600
EC48-  90 DD       BCC   $EC27
EC4A-  60          RTS                  ; RTS nur für Fehler

EC4B-  A9 4A       LDA   #$4A           ; ERR=4A: INCOMPATIBLE FILE FORMAT
EC4D-  20 09 BF    JSR   $BF09          ; SET SYSERR, RTS zu $D078
```

– Setzt „FILE ALTERED" („FLUSH always")

Xref: $E58A – JP vom Exit des WRITE Transfer-Loops
$E860 – JP vom Setzen eines neuen EOF im FCB-Eintrag (SET EOF)

```
EC50-  48          PHA
EC51-  98          TYA
EC52-  48          PHA
EC53-  AC 52 F0    LDY   $F052          ; FCB-Index
EC56-  B9 1C F3    LDA   $F31C,Y        ; Byte $1C: „File altered"-Flag
EC59-  09 80       ORA   #$80             für FLUSH („FLUSH always")
EC5B-  99 1C F3    STA   $F31C,Y
EC5E-  68          PLA
EC5F-  A8          TAY
EC60-  68          PLA
EC61-  60          RTS
```

**– REMOVE BLOCKS**

Xref: $E8EF – SET EOF: $F084..$F086 auf neuen EOF gesetzt
$EBD2 – DESTROY: $F084..$F086 auf 00 02 00 gesetzt (nur erster Block bleibt)

```
EC62-  AD 81 F0    LDA   $F081          ; Storage Type
EC65-  C9 20       CMP   #$20
EC67-  90 0B       BCC   $EC74
EC69-  C9 30       CMP   #$30
EC6B-  90 0A       BCC   $EC77
EC6D-  C9 40       CMP   #$40
EC6F-  90 09       BCC   $EC7A
EC71-  20 0C BF    JSR   $BF0C          ; SYSTEM DEATH, s.„Spezielles"
EC74-  4C 46 ED    JMP   $ED46          ; REMOVE BLOCKS für Storage Type 1
EC77-  4C 0D ED    JMP   $ED0D          ; REMOVE BLOCKS für Storage Type 2
```

– REMOVE BLOCKS für Storage Type 3

```
EC7A-  A9 80      LDA  #$80        ; Start vom höchstmöglichen Block her
EC7C-  8D 88 F0   STA  $F088       ; Index im MASTER
EC7F-  20 71 ED   JSR  $ED71       ; liest „ersten Block im File" – hier MASTER
EC82-  B0 5B      BCS  $ECDF
EC84-  AC 88 F0   LDY  $F088
EC87-  CC 84 F0   CPY  $F084       ; UNTERGRENZE IM MASTER-BLOCK
EC8A-  F0 54      BEQ  $ECE0       ; ist erreicht => REMOVE BLOCKS Type 2

EC8C-  A2 07      LDX  #$07        ; es werden bis zu 8 existierende (<> 00 00)
EC8E-  B9 00 F6   LDA  $F600,Y       Indexblocknummern auf $F08A...92 kopiert.
EC91-  9D 8A F0   STA  $F08A,X       Dieser Loop startet mit $80, d.h. bei
EC94-  19 00 F7   ORA  $F700,Y       einem File < 16 MByte werden erst einmal
EC97-  F0 09      BEQ  $ECA2         alle Indexblocknummern, die nie belegt
EC99-  B9 00 F7   LDA  $F700,Y       waren, übersprungen.
EC9C-  9D 92 F0   STA  $F092,X
EC9F-  CA         DEX
ECA0-  30 12      BMI  $ECB4
ECA2-  88         DEY
ECA3-  CC 84 F0   CPY  $F084       ; Untergrenze erreicht?
ECA6-  D0 E6      BNE  $EC8E
ECA8-  C8         INY
ECA9-  A9 00      LDA  #$00        ; Ja, Auffüllen der restlichen Tabelle mit
ECAB-  9D 8A F0   STA  $F08A,X       Nullen (= Tabellenende)
ECAE-  9D 92 F0   STA  $F092,X
ECB1-  CA         DEX
ECB2-  10 F7      BPL  $ECAB
```

– Loop durch maximal 8 kopierte Indexblocknummern

```
ECB4-  88         DEY
ECB5-  8C 88 F0   STY  $F088       ; Index für nächste Kopie von 8 Indexblocks
ECB8-  A2 07      LDX  #$07
ECBA-  8E 89 F0   STX  $F089       ; Herunterzähler
ECBD-  BD 8A F0   LDA  $F08A,X
ECC0-  85 46      STA  $46         ; Setzen der Blocknummer
ECC2-  1D 92 F0   ORA  $F092,X       und Test auf Tabellenende
ECC5-  F0 B8      BEQ  $EC7F       ; Tabellenende erreicht
ECC7-  BD 92 F0   LDA  $F092,X
ECCA-  85 47      STA  $47
ECCC-  20 E1 DD   JSR  $DDE1       ; Lesen e. Indexblocks auf $F600: MASTER-
ECCF-  B0 0E      BCS  $ECDF         Block wird dabei im Speicher überschrieben
ECD1-  20 A0 ED   JSR  $EDA0       ; entfernt alle Datenblocks dieser Indizes,
ECD4-  B0 09      BCS  $ECDF         die Indexblocks selber bleiben belegt!
ECD6-  AE 89 F0   LDX  $F089       ; Herunterzähler
ECD9-  CA         DEX
ECDA-  10 DE      BPL  $ECBA       ; Loop
ECDC-  30 A1      BMI  $EC7F       ; nächste 8 Indexblocks

ECDE-  18         CLC              ; kein Exit für $EC7A..$ECDC!
ECDF-  60         RTS
```

– Entfernung der Indexblocks aus dem MASTER-Block

Wenn keine weiteren ganzen Indexblocks mehr zu entfernen sind, wird der MASTER-Block ein weiteres Mal geladen ($EC7F) und hierher gesprungen ($EC8A).

```
ECE0-  AC 84 F0   LDY  $F084       ; Untergrenze im MASTER
ECE3-  C8         INY              ; + 1, letzter Indexblock bleibt!
ECE4-  20 A2 ED   JSR  $EDA2       ; entfernt die Indexblocks selber ab der
ECE7-  B0 F6      BCS  $ECDF         Untergrenze aus VBM und MASTER-Block
ECE9-  20 DD DD   JSR  $DDDD       ; Rewrite des MASTER-Blocks
ECEC-  B0 F1      BCS  $ECDF
ECEE-  AC 84 F0   LDY  $F084       ; Sprung, wenn Untergrenze = 00, d.h.
```

```
ECF1-   F0 15       BEQ   $ED08          alle Indexblocks entfernt
ECF3-   B9 00 F6    LDA   $F600,Y       ; sonst wird der jetzt letzte Indexblock
ECF6-   85 46       STA   $46             gesetzt und geladen
ECF8-   19 00 F7    ORA   $F700,Y
ECFB-   F0 E1       BEQ   $ECDE         ; „sparse" Indexblock: CLC/RTS
ECFD-   B9 00 F7    LDA   $F700,Y
ED00-   85 47       STA   $47
ED02-   20 E1 DD    JSR   $DDE1         ; liest den jetzt letzten Indexblock
ED05-   90 0B       BCC   $ED12
ED07-   60          RTS

ED08-   20 7E ED    JSR   $ED7E         ; MASTER-Block ist leer: DEALLOCATE des
ED0B-   B0 D2       BCS   $ECDF           MASTER-Blocks selber, Type-Herabsetzung
```

– REMOVE BLOCKS für Storage Type 2

Xref: $EC72 – JP – File hat Storage Type 2
$ED08 – MASTER Block entfernt, File hat jetzt Type 2

```
ED0D-   20 71 ED    JSR   $ED71         ; liest den „ersten Block im File"
ED10-   B0 CD       BCS   $ECDF
```

Xref: $ED05 – SET EOF: „erster Block" ist immer noch MASTER, letzter Index geladen

```
ED12-   AC 85 F0    LDY   $F085         ; UNTERGRENZE IM INDEXBLOCK
ED15-   C8          INY                 ; letzter Datenblock bleibt bestehen!
ED16-   F0 0A       BEQ   $ED22         ; Sprung bei Untergrenze = $FF, kein REMOVE
ED18-   20 A2 ED    JSR   $EDA2         ; entfernt alle Datenblocks ab Untergrenze
ED1B-   B0 C2       BCS   $ECDF           aus dem Indexblock (bzw. aus der VBM)
ED1D-   20 DD DD    JSR   $DDDD         ; Rewrite des Indexblocks
ED20-   B0 BD       BCS   $ECDF
ED22-   AC 85 F0    LDY   $F085         ; war Untergrenze = 00, d.h. Inhalt
ED25-   F0 15       BEQ   $ED3C           komplett entfernt?
ED27-   B9 00 F6    LDA   $F600,Y       ; nein, Setzen des jetzt letzten Datenblocks
ED2A-   85 46       STA   $46
ED2C-   19 00 F7    ORA   $F700,Y
ED2F-   F0 AD       BEQ   $ECDE         ; „sparse" Datenblock: CLC/RTS
ED31-   B9 00 F7    LDA   $F700,Y
ED34-   85 47       STA   $47
ED36-   20 E1 DD    JSR   $DDE1         ; liest den Datenblock auf $F600
ED39-   90 10       BCC   $ED4B
ED3B-   60          RTS
```

– Inhalt des Indexblocks komplett entfernt:

```
ED3C-   AD 84 F0    LDA   $F084         ; Untergrenze im MASTER ebenfalls 00 ?
ED3F-   D0 E6       BNE   $ED27         ; nein, File hat Type 3!
ED41-   20 7E ED    JSR   $ED7E         ; ja, Entfernung des Indexblocks aus der
ED44-   B0 2A       BCS   $ED70           VBM und Herabsetzen des Storage Type
```

– REMOVE BLOCKS für Storage Type 1

Xref: $EC74 – JP: File hat Storage Type 1
$ED44 – File hat jetzt Storage Type 1 – einziger Indexblock komplett entfernt

```
ED46-   20 71 ED    JSR   $ED71         ; liest „ersten Block im File"
ED49-   B0 25       BCS   $ED70
```

Xref: $ED05 – File hat immer noch einen höheren Type – letzter Datenblock geladen

```
ED4B-  AC 87 F0    LDY   $F087        ; $F086/87: Untergrenze für Löschung
ED4E-  F0 06       BEQ   $ED56          innerhalb dieses Datenblocks. Für
ED50-  88          DEY                  DESTROY auf $200 gesetzt, d.h. keine
ED51-  D0 1C       BNE   $ED6F          Löschung, der Datenblock wird komplett
ED53-  AC 86 F0    LDY   $F086          entfernt. Für SET EOF beliebige Werte
ED56-  A9 00       LDA   #$00           im Bereich 00 00 bis 01 FF
ED58-  99 00 F7    STA   $F700,Y
ED5B-  C8          INY
ED5C-  D0 FA       BNE   $ED58
ED5E-  AC 87 F0    LDY   $F087
ED61-  D0 09       BNE   $ED6C
ED63-  AC 86 F0    LDY   $F086        ; wenn $F086/87 einen Wert <> $200 haben,
ED66-  99 00 F6    STA   $F600,Y        wird der Datenblock nach Löschung
ED69-  C8          INY                  im Speicher zurückgeschrieben
ED6A-  D0 FA       BNE   $ED66
ED6C-  4C DD DD    JMP   $DDDD        ; schreibt den Block zurück
ED6F-  18          CLC                ; ansonsten schlicht RTS
ED70-  60          RTS
```

– Liest den „ersten Block im File" auf $F600/$F700

Xref: $EC7F – Lesen des MASTER-Blocks
$ED0D – Lesen des (einzigen) Indexblocks
$ED46 – Lesen des (einzigen) Datenblocks

```
ED71-  AD 7F F0    LDA   $F07F        ; Nummer des „ersten Blocks im File",
ED74-  85 46       STA   $46            von DESTROY/SET EOF aus dem File Info
ED76-  AD 80 F0    LDA   $F080          kopiert und bei Type-Herabsetzung
ED79-  85 47       STA   $47            verändert
ED7B-  4C E1 DD    JMP   $DDE1        ; RTS von da aus
```

– Entfernt den „ersten Block" und erniedrigt den Storage Type des Files

Xref: $ED08 – Entfernung des MASTER, „erster Block" ist danach (einziger) Indexblock
$ED41 – Entfernung des einzigen Indexblocks, „erster Block" danach Datenblock

```
ED7E-  AE 80 F0    LDX   $F080        ; Nummer des „ersten Blocks im File"
ED81-  AD 7F F0    LDA   $F07F        ; Lobyte
ED84-  20 27 DC    JSR   $DC27        ; DEALLOCATE BLOCK
ED87-  B0 16       BCS   $ED9F
ED89-  AD 00 F6    LDA   $F600        ; enthält die Blocknummer des jeweils
ED8C-  8D 7F F0    STA   $F07F          „nächstniedrigeren" Block-Typs und
ED8F-  AD 00 F7    LDA   $F700           ist als einzige Blockno bis jetzt
ED92-  8D 80 F0    STA   $F080           nicht entfernt worden. Wird als neuer
ED95-  38          SEC                   „erster Block" gesetzt.
ED96-  AD 81 F0    LDA   $F081
ED99-  E9 10       SBC   #$10         ; Storage Type - $10, d.h. Verminderung
ED9B-  8D 81 F0    STA   $F081          um 1
ED9E-  18          CLC
ED9F-  60          RTS
```

– Entfernung von Blocks, auf die ein MASTER/Indexblock zeigt

Xref: $ECD1 – Entfernung aller Datenblocks aus einem Indexblock

```
EDA0-  A0 00      LDY   #$00
```

Xref: $ECE4 – Entfernung von Indexblock aus dem MASTER ab Untergrenze
$ED18 – Entfernung von Datenblocks aus einem Indexblock ab Untergrenze

```
EDA2-  A5 46      LDA   $46
EDA4-  48         PHA
EDA5-  A5 47      LDA   $47          ; die momentane Blocknummer ist die
EDA7-  48         PHA                  des MASTER/Indexblocks, speichern
EDA8-  8C 5D F0   STY   $F05D        ; Untergrenze
EDAB-  B9 00 F6   LDA   $F600,Y
EDAE-  D9 00 F7   CMP   $F700,Y
EDB1-  D0 04      BNE   $EDB7
EDB3-  C9 00      CMP   #$00
EDB5-  F0 13      BEQ   $EDCA        ; Blockno = 00 00, kein DEALLOCATE nötig
EDB7-  BE 00 F7   LDX   $F700,Y
EDBA-  20 27 DC   JSR   $DC27        ; DEALLOCATE BLOCK - liest/schreibt evtl.
EDBD-  B0 0F      BCS   $EDCE          VBM-Blocks (extents)
EDBF-  AC 5D F0   LDY   $F05D
EDC2-  A9 00      LDA   #$00
EDC4-  99 00 F6   STA   $F600,Y      ; Block aus der VBM entfernt - jetzt
EDC7-  99 00 F7   STA   $F700,Y        Löschen des Eintrags im MASTER/Index
EDCA-  C8         INY
EDCB-  D0 DB      BNE   $EDA8        ; Loop bis Blockende
EDCD-  18         CLC
EDCE-  AA         TAX                ; 00 oder ERR# im Fehlerfall
EDCF-  68         PLA
EDD0-  85 47      STA   $47          ; Restore der Blocknummer
EDD2-  68         PLA
EDD3-  85 46      STA   $46
EDD5-  8A         TXA
EDD6-  60         RTS                ; mit A <> 00 im Fehlerfall, kein Carry!
```

– Setzt einen Filepuffer, Startadresse aus dem „user"-Parmblock

Xref: $E14A – OPEN: Startadresse des Filepuffers sind Bytes $04/$05 im Parmblock

```
EDD7-  A0 04      LDY   #$04
```

Xref: $EEA9 – SET BUFFER: Neuer Pufferstart sind Bytes $03/$04 im Parmblock

```
EDD9-  B1 40      LDA   ($40),Y      ; Pufferstartadresse hi
EDDB-  AA         TAX
EDDC-  C9 08      CMP   #$08         ; muß im Bereich $0800-$BBFF liegen
EDDE-  90 42      BCC   $EE22
EDE0-  C9 BC      CMP   #$BC
EDE2-  B0 3E      BCS   $EE22
EDE4-  85 4B      STA   $4B
EDE6-  88         DEY
EDE7-  B1 40      LDA   ($40),Y
EDE9-  85 4A      STA   $4A          ; $004A/4B: Puffer-Startadresse
EDEB-  D0 35      BNE   $EE22
EDED-  E8         INX
EDEE-  E8         INX
```

```
EDEF-   E8          INX
EDF0-   E8          INX                 ; Pufferlänge ist $0400 Bytes, d.h.
EDF1-   CA          DEX                   Endadresse ist $300 Bytes über Start
EDF2-   20 5D EE    JSR   $EE5D         ; berechnet Y-Index und Bit Mask
EDF5-   39 58 BF    AND   $BF58,Y       ; System Bit Map
EDF8-   D0 28       BNE   $EE22         ; Speicherseite nicht frei
EDFA-   E4 4B       CPX   $4B           ; Loop abwärts für alle vier Seiten
EDFC-   D0 F3       BNE   $EDF1
EDFE-   E8          INX
EDFF-   E8          INX
EE00-   E8          INX
EE01-   E8          INX
EE02-   CA          DEX
EE03-   20 5D EE    JSR   $EE5D         ; derselbe Loop noch einmal, diesmal wird
EE06-   19 58 BF    ORA   $BF58,Y         das entsprechende Bit in der System
EE09-   99 58 BF    STA   $BF58,Y         Bit Map belegt
EE0C-   E4 4B       CPX   $4B
EE0E-   D0 F2       BNE   $EE02
EE10-   AC 52 F0    LDY   $F052         ; FCB-Index für das File
EE13-   B9 00 F3    LDA   $F300,Y       ; Reference Nummer
EE16-   0A          ASL
EE17-   99 0B F3    STA   $F30B,Y       ; * 2 ist Index in der BUFFER TABLE
EE1A-   AA          TAX
EE1B-   A5 4B       LDA   $4B
EE1D-   9D 6F BF    STA   $BF6F,X       ; Pufferstartadresse => System Global Page
EE20-   18          CLC
EE21-   60          RTS

EE22-   A9 56       LDA   #$56          ; ERR#56: BAD BUFFER ADDRESS
EE24-   38          SEC
EE25-   60          RTS
```

– Kopiert eine Puffer-Startadresse von der SGP auf $F09D/9E

Xref: $D3E3 – SET FILE DATA
$EE34 – REMOVE BUFFER: OPEN Error Exit und CLOSE

```
EE26-   AA          TAX                 ; Aufruf mit File Refno * 2 im Akku
EE27-   BD 6E BF    LDA   $BF6E,X       ; BUFFER TABLE - Lobyte
EE2A-   8D 9D F0    STA   $F09D
EE2D-   BD 6F BF    LDA   $BF6F,X       ; Hibytes
EE30-   8D 9E F0    STA   $F09E
EE33-   60          RTS
```

**– DEALLOCATE FILEBUFFER**
(aus der System Bit Map und aus der BUFFER TABLE)

Xref: $E17C – OPEN Error Exit
$E6BD – CLOSE

```
EE34-   20 26 EE    JSR   $EE26         ; kopiert die Puffer-Startadresse
EE37-   09 00       ORA   #$00
EE39-   F0 20       BEQ   $EE5B         ; kein Puffer existent: CLC/RTS
EE3B-   A9 00       LDA   #$00
EE3D-   9D 6F BF    STA   $BF6F,X       ; sonst wird die Startadresse in der
EE40-   9D 6E BF    STA   $BF6E,X         BUFFER TABLE gelöscht
```

Xref: $EEB8 – SET BUF: Startadresse in der BUFFER TABLE enthält bereits neuen Puffer

```
EE43-  AE 9E F0    LDX   $F09E          ; Puffer-Startadresse hi
EE46-  E8          INX
EE47-  E8          INX
EE48-  E8          INX
EE49-  E8          INX                  ; Pufferlänge ist $0400 Byte, d.h. das Ende
EE4A-  CA          DEX                    des Puffers liegt 3 Speicherseiten höher
EE4B-  20 5D EE    JSR   $EE5D          ; berechnet Bit Mask und Y-Index
EE4E-  49 FF       EOR   #$FF           ; Complement
EE50-  39 58 BF    AND   $BF58,Y        ; das entsprechende Bit in der System
EE53-  99 58 BF    STA   $BF58,Y          BIT MAP wird wieder gelöscht, d.h.
EE56-  EC 9E F0    CPX   $F09E            die Speicherseite wieder freigegeben
EE59-  D0 EF       BNE   $EE4A          ; Loop abwärts bis zum Pufferstart
EE5B-  18          CLC
EE5C-  60          RTS
```

– Berechnet den Y-Index und die Bit Mask für eine Speicherseite in der SBM

Xref: $EDF2 – Setzen des Puffers mit „user"-Parametern: Check mit System Bit Map
$EE03 – Setzen des Puffers mit „user"-Parametern: Belegung in SBM
$EE4B – DEALLOCATE BUFFER
$EE8A – Test eines beliebigen Bereichs gegen SBM

```
EE5D-  8A          TXA                  ; Aufruf mit Hibyte des Adresse in X-Reg
EE5E-  29 07       AND   #$07
EE60-  A8          TAY
EE61-  B9 C0 EF    LDA   $EFC0,Y        ; Bit Mask: $80/$40/$20/$10/$08/$04/$02/$01
EE64-  48          PHA
EE65-  8A          TXA
EE66-  4A          LSR                  ; Speicherseiten-Nummer durch 8 ist Index
EE67-  4A          LSR                    in der SBM - jedes SBM-Byte enthält
EE68-  4A          LSR                    8 Speicherseiten
EE69-  A8          TAY
EE6A-  68          PLA                  ; Bit Mask
EE6B-  60          RTS
```

– Check eines beliebigen Speicherbereichs gegen die System Bit Map

Xref: $D398 – GET PREFIX: Test des Zielpuffers für das über RTS erhaltene PREFIX
$D447 – ONLINE: Test des Zielpuffers für die Volume-Namen
$E23C – READ: Test des Zielbereichs

```
EE6C-  A5 4F       LDA   $4F            ; Aufruf mit $004E/4F = Startadresse des
EE6E-  C9 02       CMP   #$02             Zielpuffers, muß > $200 sein
EE70-  90 B0       BCC   $EE22
EE72-  AE 9B F0    LDX   $F09B          ; Pufferlänge in $F09A/9B
EE75-  AD 9A F0    LDA   $F09A          ; wenn die Länge = $xx00 oder $xx01 ist,
EE78-  E9 01       SBC   #$01             wird eine Speicherseite weniger
EE7A-  B0 01       BCS   $EE7D            benötigt
EE7C-  CA          DEX
EE7D-  18          CLC

EE7E-  65 4E       ADC   $4E
EE80-  8A          TXA
EE81-  65 4F       ADC   $4F
EE83-  AA          TAX
EE84-  E0 BF       CPX   #$BF           ; Pufferende größer $BEFF?
```

```
EE86-  B0 9A     BCS  $EE22        ; ja, BAD BUFFER ADDRESS
EE88-  E8        INX
```

Xref: $D0C9 – BLOCKREAD/WRITE: $004E/4F und X (Herunterzähler von Obergrenze) bereits gesetzt, kein Test auf > $BFFF (s.„Spezielles")

```
EE89-  CA        DEX
EE8A-  20 5D EE  JSR  $EE5D        ; berechnet Y-Index und Bit Mask
EE8D-  39 58 BF  AND  $BF58,Y      ; System Bit Map
EE90-  D0 90     BNE  $EE22        ; Seite belegt, BAD BUFFER ADDRESS
EE92-  E4 4F     CPX  $4F
EE94-  D0 F3     BNE  $EE89        ; Loop abwärts, bis die Startadresse
EE96-  18        CLC                 wieder erreicht ist
EE97-  60        RTS
```

**$D3 – GET BUFFER COMMAND**; File muß OPEN sein – Set File Data vorher

```
EE98-  A0 02     LDY  #$02
EE9A-  AD 9D F0  LDA  $F09D        ; $F09D/9E: Startadresse des Filepuffers,
EE9D-  91 40     STA  ($40),Y        gesetzt durch Set File Data,
EE9F-  C8        INY
EEA0-  AD 9E F0  LDA  $F09E        ; wird in den Parameterblock des
EEA3-  91 40     STA  ($40),Y        „user"-Programms kopiert.
EEA5-  18        CLC
EEA6-  60        RTS               ; => $D078
```

**$D2 – SET BUFFER COMMAND**; File muß OPEN sein – Set File Data vorher

```
EEA7-  A0 03     LDY  #$03
EEA9-  20 D9 ED  JSR  $EDD9        ; holt die neue Puffer-Startadresse aus dem
EEAC-  B0 20     BCS  $EECE          „user"-Parmblock und checkt mit SBM, der
EEAE-  AD 9E F0  LDA  $F09E          neue Puffer wird dabei belegt und in der
EEB1-  85 4F     STA  $4F            BUFFER TABLE eingetragen
EEB3-  AD 9D F0  LDA  $F09D
EEB6-  85 4E     STA  $4E
EEB8-  20 43 EE  JSR  $EE43        ; DEALLOCATE FILEBUFFER – alter Filepuffer,
EEBB-  A0 00     LDY  #$00           2.Einsprung ohne Löschen der Startadresse
EEBD-  A2 03     LDX  #$03           in der BUFFER TABLE
EEBF-  B1 4E     LDA  ($4E),Y
EEC1-  91 4A     STA  ($4A),Y      ; der Pufferinhalt wird in den neuen
EEC3-  C8        INY                 Filepuffer umkopiert
EEC4-  D0 F9     BNE  $EEBF
EEC6-  E6 4B     INC  $4B
EEC8-  E6 4F     INC  $4F
EECA-  CA        DEX
EECB-  10 F2     BPL  $EEBF        ; Loop für $400 Byte
EECD-  18        CLC
EECE-  60        RTS               ; => $D078
```

**$65 – REBOOT COMMAND HANDLER**; keine anderen Routinen vorher

```
EECF-  AD 83 C0  LDA  $C083        ; BANKSELECT der Bank 2
EED2-  AD 83 C0  LDA  $C083        ; + WRITE ENABLE (unnötig)
EED5-  A5 00     LDA  $00
EED7-  48        PHA
EED8-  A5 01     LDA  $01          ; die Speicherstellen $0000-$0003 werden
EEDA-  48        PHA                 zwischengespeichert
EEDB-  A5 02     LDA  $02
```

```
EEDD-   48          PHA
EEDE-   A5 03       LDA   $03
EEE0-   48          PHA
EEE1-   A9 10       LDA   #$10          ; Startadresse: $1000
EEE3-   85 03       STA   $03
EEE5-   A9 D1       LDA   #$D1          ; REBOOT steht in der Bank 2 ab $D100
EEE7-   85 01       STA   $01
EEE9-   A9 00       LDA   #$00
EEEB-   85 00       STA   $00
EEED-   85 02       STA   $02
EEEF-   A8          TAY
EEF0-   A2 03       LDX   #$03          ; REBOOT ist $300 Bytes lang
EEF2-   88          DEY
EEF3-   B1 00       LDA   ($00),Y
EEF5-   91 02       STA   ($02),Y       ; und wird schlicht umkopiert
EEF7-   98          TYA
EEF8-   D0 F8       BNE   $EEF2
EEFA-   E6 01       INC   $01
EEFC-   E6 03       INC   $03
EEFE-   CA          DEX
EEFF-   D0 F1       BNE   $EEF2         ; Loop für $300 Bytes
EF01-   68          PLA
EF02-   85 03       STA   $03
EF04-   68          PLA
EF05-   85 02       STA   $02           ; Restore der Speicherstellen $00-$03:
EF07-   68          PLA                   das ist zwar sehr korrekt, ein Grund
EF08-   85 01       STA   $01             für diese Zwischenspeicherung ist
EF0A-   68          PLA                   allerdings beim momentanen REBOOT
EF0B-   85 00       STA   $00            nicht gegeben.
EF0D-   AD 8B C0    LDA   $C08B
EF10-   AD 8B C0    LDA   $C08B         ; BANKSELECT Bank 1 (+ WRITE ENABLE)
EF13-   A9 00       LDA   #$00
EF15-   8D F2 03    STA   $03F2
EF18-   A9 10       LDA   #$10          ; der RESET-Vektor wird auf $1000, also
EF1A-   8D F3 03    STA   $03F3           auf den Anfang des kopierten REBOOT
EF1D-   49 A5       EOR   #$A5            gesetzt.
EF1F-   8D F4 03    STA   $03F4
EF22-   4C 00 10    JMP   $1000         ; Exec REBOOT
```

### - Tabellen

```
EF25- D3 00 00 00 40 41 00 00  80 81 82 65 C0 C1 C2 C3  ; COMMAND-Bytes
      C4 C5 C6 C7 C8 C9 CA CB  CC CD CE CF 00 D0 D1 D2

EF45- 02 FF FF FF 02 01 FF FF  03 03 00 04 07 01 02 07  ; Anzahl der Parameter
      0A 02 01 01 03 03 04 04  01 01 02 02 FF 02 02 02    für die COMMANDS

EF65- F7 D4 50 EB 0B EA D4 E9  6F E9 2A D4 2A D3 82 D3  ; Startadressen für die
      B9 E0 5D E9 F7 E1 7C E4  77 E6 E8 E6 1E DE 08 DE    COMMANDS $C0 bis $D3

      07 E8 4B E9 A7 EE 98 EE

EF8D- A0 A1 A2 A3 84 05 06 07  88 49 4A 4B 2C 2D 4E 4F  ; $C0-$D3: „Vorhers"
      50 51 52 53                                         (Bit 7-5) und Indizes
                                                          für $EF65...

EFA1- 01 00 00 02 00     ; Anhängen eines Subdir-Blocks: BLOCKS-USED + 1, EOF + $200
```

```
------------------- In einem SubDIR Key kopiert---------
EFA6- 75 48 55 53 54 4F 4E  21                          ; uHUSTON
EFAE- 00 00             ; Für Check mit Backward Reference: (00 00 = Key Block)
EFB0- 00 00             ; MLI Version: min-max
EFB2- C3 27 0D          ; ACCESS, Eintragslänge, Anzahl Einträge per Block
EFB5- 00 00 00
-------------------------------------------------------

EFB8- 0F 02 00 04 00 08 00 00   ; FIND FILE: „gefälschtes" File Info für VolDIR:
                                  Storage Type $0F (nicht benutzt),
                                  „erster Block" 00 02, Blocks-used 00 04, EOF $0800

EFC0- 80 40 20 10 08 04 02 01           ; Bit Mask für System Bit Map / VBM

EFC8- 0C 0D 18 19 15 16 17 1E 10 1F 20  ; Adress-Übersetzung für FILE INFO
      80 93 94 21 22 23 24 18 19 1A 1B    MSB gesetzt: Überspringen des „Null_Field"
```

– SYSTEM DEATH-Text. Wird direkt in den Bildspeicher kopiert

```
EFDF- C9 CE D3 C5 D2 D4 A0 D3  D9 D3 D4 C5 CD A0 C4 C9    INSERT SYSTEM DI
EFEF- D3 CB A0 C1 CE C4 A0 D2  C5 D3 D4 C1 D2 D4 A0 A0    SK AND RESTART
EFFF- A0
F000- A0 A0 A0 A0 A0 A0 ; belegt die ersten 5 Bytes des Variablen-Bereichs
```

# 1.7. Die Variablen des MLI

**– der Speicherbereich $F000-$FFFF ohne die DISKII-RWTB**

Die Variablen und temporären Speicherplätze, die vom MLI benutzt werden, befinden sich in der Speicherseite $F000. Einige Speicherstellen werden überhaupt nicht benutzt, andere – speziell die „Scratches" (temporär benutzte Zwischenspeicher) – haben verschiedene Zwekke und sind deshalb nicht eindeutig definierbar. Ihre Bedeutung ist von der Routine abhängig, die diese Speicherstellen benutzt.
Wenn Speicherstellen zu Zweier- und Dreiergruppen zusamengefaßt werden (d.h. für 16- und 24-Bit-Werte), ist die Reihenfolge immer Lo-Hi, d.h. das niederwertigste Byte hat die niedrigere Speicheradresse. Eine Ausnahme gibt es auch hier: die Speicherstellen ($F081-$F087), die von SET EOF und DESTROY für die Routine REMOVE BLOCKS gesetzt werden, gehen etwas „durcheinander", was die „Wertigkeit" betrifft.

Die RWTB für die DISKII hat (korrekterweise) ihren eigenen Variablenbereich!

$F000-$F005 Belegt mit Leerzeichen von „INSERT SYSTEM DISK AND RESTART "

Befindet sich der gesuchte Eintrag in einer Subdirectory, werden die folgenden Speicherstellen belegt, ansonsten bekommen sie den Wert 00:

$F006/F007 Blocknummer der Directory, wo die Subdirectory als Eintrag steht („Parent Block").

$F008 Eintragsnummer innerhalb dieses Blocks.
$F009 Länge eines Eintrags innerhalb dieses Blocks.

Für alle gelesenen Directories, egal ob Volume oder Subdirectory, wird belegt:
$F00A..0D CREATION DATE & TIME, bzw. FORMAT DATE & TIME, wenn es sich bei der gelesenen Directory um eine Volume Directory handelt. Diese Speicherstellen werden nirgendwo wieder gelesen.
$F00E/0F VERSION – diese Speicherstellen werden nirgenwo wieder gelesen.
$F010 ACCESS der Directory. Wird kopiert und bei ENTRY & UPDATE wieder zurückübertragen. Eine Veränderung oder Prüfung findet nicht statt.
$F011 Länge eines Eintrags in dieser Directory ($27).
$F012 Anzahl der File-Einträge pro Block ($0D).
$F013/14 Anzahl der aktiven, d.h. nicht freien/gelöschten Einträge innerhalb sämtlicher Blocks dieser Directory.

Die folgenden Informationen werden einmal (beim „Login" via internes ONLINE) aus dem Key Block der Volume Directory gelesen, in den VCB-Eintrag kopiert und von da aus jeweils neu gesetzt:
$F015/16 Blocknummer des ersten VBM-Blocks (00 06).
$F017/18 Gesamtzahl der Blocks auf dem Volume (DiskII: $0118).
$F019 Unitnummer.

Für den Eintrag selber existieren folgende Speicherstellen:
$F01A/1B Blocknummer des Key Blocks der Directory, in der der Eintrag steht.
$F01C/1D Blocknummer des Blocks der Directory, in dem der Eintrag steht.
$F01E Eintragsnummer innerhalb dieses Blocks.

$F01F Storage Type und Namenslänge des Eintrags ($:Stype; $xX:Länge).
$F020..2E Der Name selber in ASCII, aufgefüllt mit 00 (nicht Leerzeichen).
$F02F File Type. Wird von MLI nur kopiert, nie verändert oder bearbeitet/berücksichtigt. Ausnahme: Laden des „.SYSTEM"-Programms in REBOOT oder nach dem Boot von ProDOS. Hier wird nach einem File mit File Type $FF (SYS) gesucht.
$F030/31 Nummer des ersten Blocks im File (kann Master/Index/Datenblock oder der Key Block einer (Sub)directory sein).
$F032/33 Anzahl der vom File belegten Blocks (Blocks-used) incl. Indexblocks etc.
$F034/35/36 EOF des Files, nicht unbedingt identisch mit Blocks-used.
$F037..3A CREATION DATE & TIME – nur von CREATE gesetzt.
$F03B/3C VERSION min/max (00 00) – wird bei OPEN geprüft.
$F03D ACCESS des Files – wird im Gegensatz zu ACCESS der Directory geprüft.
$F03E/3F AUX-INFO. Wird GET FILE für Volume Directories ausgeführt, landen hier „Länge eines Eintrags"/„Einträge pro Block" und werden entweder nicht berücksichtigt (READ) oder sofort überschrieben (GET FILE INFO). Für Datenfiles steht hier z.B. die Startadresse im Speicher (Maschinenprogramme, BASIC...).

| | |
|---|---|
| $F040..43 | MODIFICATION DATE & TIME |
| $F044/45 | Nummer des Key Blocks der Directory, in der dieser Eintrag steht. Diese Information wird bei CREATE angehängt und immer nur wieder kopiert, wenn der Eintrag (ENTRY & UPDATE) neu geschrieben wird. Berücksichtigt oder ausgewertet wird die Information nie, das Programm holt sich die entsprechende Blocknummer aus $F01A/1B. |

Der Rest der Speicherseite $F000 sind interne Variablen des MLI:

| | |
|---|---|
| $F046 | – in SET MARK: Differenz zwischen „Position jetzt" und „Position gewünscht" in Blocks * 2. Für SET MARK in Directories ist das die Anzahl der Blocks/2, die übersprungen werden muß.<br>– in FLUSH: Temporär für das Addieren des Storage Type zur Namenslänge des Files verwendet. |
| $F046/47 | – Zähler für die freien Blocks in „Space on Volume" ($DB66).<br>– Nummer des nächsten freien Blocks (ALLOCATE NEXT BLOCK: $DCA9). |
| $F046/47/48 | Ergebnis der Addition von „Position jetzt" und Request Count für READ und WRITE. Wird mit dem EOF verglichen. Für READ wird der Transfer-Zähler entsprechend heruntergesetzt, bei WRITE der EOF entsprechend erhöht. |
| $F049 | nicht verwendet |
| $F04A/4B/4C | Momentaner EOF des Files, kopiert aus dem FCB-Eintrag |
| $F04D/4E/4F | Alte „Position jetzt" im File. Wird von SET MARK aus dem FCB-Eintrag kopiert und vom Error Exit von WRITE verwendet. |
| $F050 | Temporär im Vergleich „Key Block – VCB-Eintrag" ($DB1C) verwendet und sonst nirgendwo („Scratch"). |
| $F051 | VCB-Index ($00/$20/$40/$60/$80..): Basis des momentan benutzten VCB-Eintrags. |
| $F052 | FCB-Index ($00/$20/$40...): Basis des momentan benutzten FCB-Eintrags. |
| $F053 | „Freier Eintrag im FCB gefunden"-Flag. 00 – kein freier Eintrag gefunden. Gesetzt von $E1A9 („Scan FCB"), benutzt von OPEN. |
| $F054/55 | „Blocks needed" – Anzahl der Blocks, die belegt werden sollen. Gesetzt vor jedem Aufruf von „Space on Volume" ($DB66). Bei der Berechnung der Anzahl der belegten Blöcke für GET FILE INFO (Volume Directories) auch als Zwischenergebnis verwendet. |

$F056 Nur innerhalb von SET MARK verwendet: Storage Type des Files, kopiert aus dem FCB-Eintrag.

$F057 – Von FIND FILE ENTRY als Zähler für die aktiven Einträge innerhalb eines Directory-Blocks verwendet.
– Für OPEN/RENAME/DESTROY: Flag in „scan FCB" ($E1A9). Gesetzt <> 0, wenn das File als FCB-Eintrag existiert, d.h. OPEN ist.

$F058/59 Kopie von $F013/14: Anzahl der aktiven Einträge. Wird in FIND FILE ENTRY beim Absuchen einer Directory nach einem bestimmten Eintrag heruntergezählt (Verifizierung der DIRECTORY STRUCTURE: Es müssen bei der Suche genauso viele Einträge gezählt werden, wie angegeben). Dient innerhalb derselben Routine auch noch als „Indikator" für CREATE.

$F05A – In FIND FILE ENTRY: (Herunter)zähler für die Einträge innerhalb eines Directory-Blocks, Kopie von $F012 (Wert: $0D).
– In SET MARK für Directories: Herunter/Heraufzähler für die Anzahl der zu überspringenden Directory-Blocks ($F046 / 2).
– Für OPEN: Nummer (nicht Index) der ersten freien FCB-Eintrags, der von „scan FCB" ($E1A9) gefunden wurde.

$F05B – „Freier Eintrag in der Directory gefunden"-Flag für CREATE, gesetzt von FIND FILE ENTRY beim Absuchen der Directory ($D7E0).
–In SPACE ON VOLUME ($DB66) am Anfang auf $FF („no Space") gesetzt. Wird danach auf den ersten VBM-Extent gesetzt, in dem ein freier Block gefunden wurde ($DC01) oder bleibt auf $FF => VOLUME FULL.
–In DEALLOCATE BLOCK ($DC27) temporär für die Bitmask ($80/$40/$20/$10/$08..).

$F05C Herunterzähler in „Space on Volume" ($DB66) für die Anzahl der möglichen VBM-Blocks.

$F05D Start- und momentaner Index in REMOVE BLOCKS für das DEALLOCATE von Blocks, auf die ein Master/Indexblock zeigt. Temporäre Variable, wird innerhalb von REMOVE BLOCKS gesetzt!

$F05E Gesamtlänge des Pathname (+ Prefix) in Copy & Verify Pathname.

$F05F PREFIX UNIT. Wird von SET PREFIX auf die entsprechende Unitnummer gesetzt, aber leider nirgendwo benutzt. Das Volume wird bei einem gesetzten Prefix im VCB gesucht und die Unitnummer aus dem gefundenen VCB-Eintrag geholt. Das ist auch ganz gut so, denn bei $D970 wird diese Speicherstelle als Scratch verwendet.

$F060/61 PREFIX DIRECTORY. Wird von SET PREFIX gesetzt und beinhaltet die Blocknummer des Key Blocks der „PREFIXten" Directory. Enthält

das Prefix nur einen Namen, haben diese Speicherstellen den Wert 0002, d.h. sie zeigen auf die Volume Directory. Im Gegensatz zu PREFIX UNIT von FIND FILE ENTRY verwendet.

$F062 – In DEALLOCATE BLOCK: Blocknummer/8, d.h. Index innerhalb der VBM-Bytes (Basisadresse).
– In ALLOCATE NEXT BLOCK: Blocknummer/8, d.h. derselbe Zweck. Wird hier allerdings nur gespeichert und nie mehr benutzt.

$F063 In ALLOCATE NEXT BLOCK: Blocknummer / $800, d.h. Speicherseite innerhalb eines VBM-Blocks. Gesetzt durch VBM-„extent" / 2 beim Laden eines VBM-Blocks und während der Suche kontinuierlich erhöht. Wird hinterher zur Berechnung der Nummer eines gefundenen freien Blocks verwendet.

$F064 – In DEALLOCATE BLOCK: Speicherseite innerhalb der VBM, d.h. Blocknummer / $800 (s. $F063 in ALLOCATE NEXT BLOCK).
– In ALLOCATE NEXT BLOCK: Speicherseite innerhalb des VBM-Blocks, dient nur als Indikator (0/1), die Blocknummer wird von $F063 bestimmt.

$F065 „VBM altered". Gesetzt auf $80, wenn der VBM-Block, der sich momentan im Speicher befindet, verändert wurde und zurückgeschrieben werden muß, bevor ein anderer VBM-Block geladen werden kann. Nach dem Zurückschreiben bekommt diese Speicherstelle den Wert $00.

$F066 „VBM unit". Nummer des Units, von dem sich gerade ein VBM-Block im Speicher befindet.

$F067/68 Blocknummer des ersten VBM-Blocks für dieses Unit. Kopie aus dem VCB-Eintrag.

$F069 „VBM Extent". Nummer des VBM-Blocks (nicht Blocknummer!), der gerade im Speicher ist. Für den ersten VBM-Block hat diese Speicherstelle den Wert 0, bei einer ProFile nimmt sie Werte zwischen 00 und 02 an (3 VBM-Blocks).

$F06A/6B/6C – „Position gewünscht" im File für SET MARK.
– SET EOF: neuer EOF, später als „Position gewünscht" für SET MARK verwendet, wenn der File verkleinert wurde.

$F06D/6E Transfer-Herunterzähler für READ und WRITE. Wird durch den Request Count gesetzt und bei READ entsprechend verkleinert, wenn READ den EOF des Files überschreiten würde.

$F06F Anzahl der kompletten Blocks, die noch übertragen werden sollen. Nur von READ (Stufen 2 und 3) verwendet.

$F070 NEWLINE BYTE – nur von READ verwendet. Kopie aus dem FCB-Eintrag.

$F071 NEWLINE MASK – nur von READ verwendet. Kopie aus dem FCB-Eintrag.

$F072 Mystery Flag für READ, Stufe 3. Wird von jedem I/O via SET MARK auf $FF gesetzt, vor Stufe 2 von READ auf $00 (s. „Spezielles").

$F073 $C0-D3 Command Handler ($D23C): „Shift"-Byte für die Routinen, die vor dem eigentlichen Command auszuführen sind. Bekommt den Wert aus der Tabelle auf $EF8D. Je nach gesetzten Bits im Bereich Bit 7-5 werden „Copy & Verify Pathname" ($D27F), „Set File Data" ($D3C5) und SET TIME ($BF06) vorher ausgeführt. Nirgendwo anders verwendet.

$F074 Internes Backup-Flag des MLI (s. „Spezielles").

$F075 In ONLINE ALL: hat ein anderes Unit denselben Volume-Namen und einen offenen File, wird dieses Flag gesetzt. Führt zu ERR#57 DUPLICATE VOLUME.

$F076 In ONLINE ALL: VCB-Index des doppelten Volumes, auf dem ein offener File gefunden wurde.

$F077 $C0-$D3 Command Handler: Index innerhalb der ersten Tabelle ($00-$13). Wird beim Aufruf hier gespeichert und nie wieder benutzt.

$F078 – Copy & Verify Pathname ($D27F): Zähler für die Länge eines Namens im Path. Mit Anfangswert $FF auch gleichzeitig als Indikator „erstes Zeichen im Namen", d.h. besondere Prüfung verwendet.
– ONLINE ALL ($D4D6): Herunterzähler für das Kopieren eines Volume-Namens in den Puffer des „user"-Programms.
– „Scan Directory block" ($D8D8): Herunterzähler für den Vergleich eines Directory-Eintrags mit dem gesuchten Namen.
– „Scan VCB" ($D999): Herunterzähler für Namensvergleich.

$F079 RENAME Files/Subdirectories: Start(index) des unterschiedlichen Namens im neuen Path.

$F07A Startindex des momentan behandelten Namens im Path. Ausnahme: ONLINE ALL: Start des nächsten Eintrags im Puffer des „user"-Programms.

$F07B Startindex des (gesamten) Path: Existiert ein System Prefix und ist der angegebene Name nicht vollständig, hat diese Speicherstelle einen Wert im Bereich $C0..$FE, bei einem vollständigen Pathname hat sie den Wert $00.

$F07C FULL NAME Flag. Hat den Wert $AE (=„/") bei einem vollständigen Namen oder den Wert $00, wenn ein System Prefix vorangestellt wird. Gesetzt von Copy & Verify Pathname.

$F07D – In Copy & Verify Pathname: Scratch für den momentanen Index im Path.
– In ONLINE ALL: Herunterzähler innerhalb der von der Global Page kopierten DEVICE TABLE.

$F07E – Zwischenspeicher für Fehler, die bei CLOSE/FLUSH ALL auftreten. Gesetzt durch die Error-Routine auf $E7ED. Wird nach dem Ende des jeweiligen ALL-Loops zurückgeladen.

SET EOF und DESTROY benutzen zusammen dieselbe Routine (REMOVE BLOCKS: $EC62). DESTROY setzt dabei die Untergrenze auf $00 02 00, d.h. es bleibt ein Datenblock vom File übrig, der dann ebenfalls komplett entfernt wird.
SET EOF setzt beliebige Werte zum Verkleinern eines Files.
REMOVE BLOCKS setzt ggf. den Storage Type des Files herunter und verändert damit die Nummer des „ersten Blocks im File".

$F07F/80 „erster Block im File" für SET EOF/DESTROY. Wird von REMOVE BLOCKS verändert, wenn der Storage Type des Files heruntergesetzt wird.

$F081 Storage Type des Files für REMOVE BLOCKS.

$F082/83 Anzahl der von REMOVE BLOCKS freigegebenen Blocks.

$F084 Untergrenze für das REMOVE innerhalb des Masterblocks.

$F085 Untergrenze für das REMOVE innerhalb eines Indexblocks.

$F086/87 Untergrenze für das REMOVE innerhalb eines Datenblocks. Ist diese Untergrenze < $200, wird innerhalb des Datenblocks gelöscht und der Datenblock danach zurückgeschrieben.

REMOVE BLOCKS benutzt zusätzlich die drei Speicherstellen $F05D und $F088/89:

$F088 Index innerhalb eines Masterblocks für die Kopie von jeweils bis zu acht existierenden Indexblocks.

$F089 Herunterzähler innerhalb dieser 8 kopierten Indexblocks.

$F08A – bis $F091: Blocknummern (low) von 8 kopierten Indexblocks für REMOVE BLOCKS.
– bis $F097: aus der Global Page kopierte DEVICE TABLE für ONLINE ALL und internes ONLINE in FIND FILE ENTRY.

$F091 – bis $F099: Blocknummern (high) von 8 kopierten Indexblocks für REMOVE BLOCKS.

$F09A/9B Länge des „user"-Puffers für GET PREFIX ($40), ONLINE ($10, ALL:$100) und READ/WRITE.

$F09C nicht benutzt

$F09D/9E Startadresse des Filepuffers, aus der Global Page kopiert. Nur von Set File Data ($D3C5) und SET/GET BUFFER benutzt.

$F09F..$F0A4 IRQ Handler: Zwischenspeicher für die Speicherstellen $FA..$FF von der Zeropage.

$F0A5..$F0AE nicht benutzt

$F0AF..$F0BE Stack Space für den Fall, daß bei einem Interrupt der Stack mehr als halbvoll (SP < $80) ist. Dabei wird $F0B5/B6 überschrieben (s. „Spezielles").

$F0B5/B6 Startadresse der Commands $C0-$D3. Wird via indirektem JSR vom Command Handler ($D23C) aufgerufen.

$F0BF..$F0FF nicht benutzt

---

$F100..$F140 Pathname, wird aus dem Parameterblock des „user"-Programms von Copy & Verify Pathname ($D27F) hierher kopiert.

$F141..$F1AB Thunderclock-Routine ($82 – SET TIME Command)

$F1AB..$F1BF Tabellen für die Jahrfeld-Umrechnung der Thunderclock.

$F1C0..$F1FF System Prefix. Das System Prefix endet immer auf $F1FF, die Speicherstelle $BF94 enthält entweder den Startindex im Bereich $C0-$FE oder den Wert $00 (kein Prefix).

---

$F200..$F300 VCB – die Einträge werden hier in absteigender Reihenfolge gemacht, d.h. der erste Eintrag, der belegt wird, hat die Basis $F2C0.
Jeder Eintrag umfaßt $20 Byte, d.h. es sind maximal 8 Einträge möglich.
Da die Adressierung grundsätzlich über den VCB-Index geschieht, folgt hier ein Beispiel-Eintrag auf $F200:

$F200: 04 Länge des folgenden Namens (hier: 4) und „besetzt"-Indikator
$F201..0F NAME in ASCII-Großbuchstaben, z.b. „DEMO"
$F210: 60 Nummer des Units, für das dieser Eintrag steht. Hier: S6,D1
$F211: 00 „OPEN File(s)": 00 – kein offener File, ansonsten $80
$F212: 01 18 Gesamtzahl der Blocks auf dem Volume (DiskII: $0118)
$F214: 01 0E Anzahl der freien Blöcke, muß via „Space on Vol" bestimmt werden.

| | |
|---|---|
| $F216: 02 00 | Blocknummer des Key Blocks der Volume Directory (00 02) |
| $F218: 00 00 | nicht benutzt |
| $F21A: 06 00 | Blocknummer des (ersten) VBM-Blocks (00 06) |
| $F21C: 00 | VBM-„Extent", hier erster Block |
| $F21D: 00 | nicht benutzt |
| $F21E: 00 | Anzahl der momentan offenen Files, hier keiner |
| $F21F: 00 | nicht benutzt. |
| | |
| $F220..$F23F | nächster („siebter") VCB-Eintrag |
| $F240..$F25F | sechster VCB-Eintrag |
| .... | |
| $F2C0..$F2FF | erster VCB-Eintrag |

---

| | |
|---|---|
| $F300..$F3FF | FCB – Im Gegensatz zum VCB werden hier die Einträge in aufsteigender Reihenfolge gemacht. Auch hier umfaßt jeder Eintrag $20 Bytes, d.h. es sind maximal 8 Einträge möglich.<br>Ein Beispiel für den ersten FCB-Eintrag: |
| | |
| $F300: 01 | Nummer des Eintrags (immer Index/$20) – „Eintrag belegt" |
| $F301: 60 | Unitnummer des Files, hier S6, D1 |
| $F302: 02 00 | Nummer des Key Blocks der Directory, in der der File verzeichnet ist (hier 0002: Volume Directory). |
| $F304: 03 00 | Nummer des Directory-Blocks, in dem der Eintrag für den File steht (hier der zweite Block der Volume Directory). |
| $F306: 01 | Eintragsnummer: Der Eintrag ist der erste innerhalb des Blocks. |
| $F307: 02 | Storage Type |
| $F308: E0 | STATUS: Sowohl Index- als auch ein Datenblock befinden sich im Filepuffer und wurden verändert, sie müssen noch zurückgeschrieben werden. „File accessed" (Bit 4) ist damit ebenfalls gesetzt. |
| $F309: 03 | ACCESS: Nur READ/WRITE werden in den FCB kopiert. Hier ist beides erlaubt. |
| $F30A: 00 | NEWLINE CHARACTER (hier nicht gesetzt). |
| $F30B: 02 | Puffer Index: das File hat den ersten in der Global Page verzeichneten Puffer. |
| $F30C: 09 00 | „erster Block im File" (00 09) |
| $F30E: 09 00 | „Indexblock im Speicher": der (einzige) Indexblock ist geladen |
| $F310: 0F 00 | „Datenblock im Speicher" (00 0F). |
| $F312: FD0700 | „Position jetzt"(00 07 FD): innerhalb des vierten Datenblocks. |
| $F315: FD0700 | EOF (00 07 FD): nach einem WRITE gerade nach hinten verschoben. |
| $F318: 00 05 | BLOCKS-USED: der File benötigt 00 05 Blocks |
| $F31A: 00 | nicht benutzt |
| $F31B: 07 | „LEVEL" – Kopie von der Global Page |
| $F31C: 80 | „File altered" – der Eintrag muß bei FLUSH zurückgeschrieben werden, sonst 00. |
| $F31D: 00 00 | nicht benutzt |

$F31F: 00 NEWLINE MASK, hier nicht gesetzt.

$F320..$F33F Zweiter FCB-Eintrag
$F340..$F35F Dritter FCB-Eintrag
.....
$F3C0..$F3FF Achter FCB-Eintrag

---

$F400..$F5FF VBM Space – dieser Speicherplatz wird nur für VBM-Blocks benutzt.

---

$F600..$F7FF Directory Space – FIND FILE ENTRY lädt Directory-Blocks hierher, REMOVE BLOCKS benutzt diesen Platz auch für Master/Index- und Datenblocks, weil u.U. (DESTROY) kein Filepuffer zur Verfügung steht.

---

$F800..$FF00 RWTB für die DiskII

$FF00..$FF69 RAMdisk – RWTB

$FF6A..$FF7F Freier Bereich, kann von RAMdisk – RWTB benutzt werden

$FF80..$FFFF IRQ, NMI und RESET-Handler für den Fall, daß eines dieser Signale auftritt, während die Language Card eingeschaltet ist.

# 1.8. Einige Programmiertricks im MLI

In der Programierung von ProDOS finden sich eine ganze Menge von „Platzsparern" und Optimierungen. Hier sollen allerdings nur zwei Punkte herausgenommen werden, die nach Meinung des Autors wirklich an der Grenze zur Undurchschaubarkeit liegen. Es handelt sich dabei um die Such-Sequenz der einzelnen Units und die Behandlung des Pathname.
Fangen wir mit dem einfacheren von beiden an:

## 1.8.1. Die Absuch-Reihenfolge der Volumes („Drive Search Order")

Die Suche bei einem internen ONLINE (d.h. wenn ein Volume nicht gefunden wurde) durchläuft folgende Schritte:
– alle Devices werden aus der DEVICE TABLE kopiert
– der Startindex wird von DEVNUM geladen
– Loop abwärts durch die Devices, d.h. die Suche beginnt mit DEVTABLE, DEVNUM und endet mit DEVTABLE,00.

Dadurch wird auch verständlich, warum DEVNUM immer die Anzahl aller DEVICES minus 1 enthält.
Innerhalb des Boot/Relocators werden die Slots der Reihe nach abgesucht, die Suche beginnt mit Slot 7 ($C700) und endet mit Slot 1 ($C100). Wird dabei ein Disk-Controller erkannt (DISKID), wird die Karte in der DEVICE TABLE eingetragen und DEVNUM (Anzahl der DEVICES) um eins erhöht. Sind auf einem Controller zwei Drives möglich, wird Drive 2 zuerst eingetragen (Routine auf $24E6 während des Boot).
Bei zwei Controllern in den Slots 5 und 6 sieht die DEVICE TABLE anschließend so aus:

```
BF32- E0        ; Slot 6, Drive 2
BF33- 60        ; Slot 6, Drive 1
BF34- D0        ; Slot 5, Drive 2
BF35- 50        ; Slot 5, Drive 1
```

Hier würde eine Suche jetzt mit Slot 5, Drive 1 beginnen. Um zu vermeiden, daß die Suche mit einen eventuell „leeren" Drive beginnt, wird nach dem Absuchen aller Slots der „Boot-Slot", also der Slot, von dem aus ProDOS geladen wurde, nach oben in der DEVICE TABLE sortiert (Routine auf $2574 während des Boot) und die DEVICE TABLE ändert sich dementsprechend.
Ein Beispiel für einen Boot von Slot 6:

```
BF32- D0        ; Slot 5, Drive 2
BF33- 50        ; Slot 5, Drive 1
BF34- D0        ; Slot 6, Drive 2
BF35- 60        ; Slot 6, Drive 1
```

Eine RAMDisk stellt ein Device dar, das immer „ready" ist, solange sie existiert. Darüberhinaus ist die Zugriffszeit extrem niedrig. Aus diesem Grund wird die RAM Disk für eine „128k"-Maschine immer als oberstes Device eingetragen – der Eintrag erfolgt nach der Slot-Absuche und dem Sortieren der DEVICE TABLE (Routine auf $28FF während des Boot).

Bei einem internen ONLINE wird noch in zwei Fälle unterschieden:
1) Allgemeine Suche ohne Vorgaben: die DEVICE TABLE wird von oben nach unten abgearbeitet, also mit der RAMDisk (falls vorhanden) zuerst, danach der „Boot"-Slot und schließlich alle anderen Slots, wobei Reihenfolge der Slot**nummern** dann wieder aufsteigend ist. Drive 1 eines Slots kommt dabei immer vor Drive 2 (wenn vorhanden).

2) Es wurde auf einem beliebigen Device ein Volume-Name erwartet, der sich aber auf diesem Device nicht mehr findet (Diskette gewechselt). In diesem Fall wird dieses Device zuerst behandelt. Erst danach geht es wie in Fall 1) weiter, mit dem Unterschied, daß dieses erste Device ausgelassen wird. Aus diesem Grund wird die DEVICE TABLE vorher kopiert – während der Suche behandelte Devices werden aus der kopierten Tabelle schlicht gelöscht.
Ein „Extrembeispiel" wäre die Suche mit folgenden Vorgaben:
– RAMDisk
– „Boot"-Slot ist Slot 6
– Im VCB steht für Slot 5,D2 ein Volume verzeichnet, das jetzt gesucht wird, dessen Diskette aber gewechselt wurde.
Als Suchreihenfolge ergibt sich bei der oben angeführten DEVICE TABLE (+RAM Disk): S5,D2 / S3,D2 (RAMDisk) / S6,D1 / S6,D2 / S5,D1.

## 1.8.2. Die Behandlung des Pathname

**1) SET PREFIX**

Der Pathname wird von Copy & Verifyh Pathname ($D27F) aus dem Parameterblock des „user"-Programms geholt und nach $F100... kopiert, die Schrägstriche zwischen den einzelnen Namen des Path werden dabei durch Längenangaben für die einzelnen Namen ersetzt. Die maximale Länge des Pathname beträgt $40 Zeichen, d.h. im Maximalfall wird der Speicherbereich $F100..$F13F belegt.
Nehmen wir als Beispiel den Pathname „/DEMO/SUBDIR". Nach der Kopie findet sich auf $F100 dann (4) „DEMO" (6) „SUBDIR" (0). Der letzte Nuller steht für die Länge des dritten Namens im Path und hat die Bedeutung „keine weiteren Namen".

Wenn dieser Name als PREFIX gesetzt wird, wird er so kopiert, daß er auf $F1FF endet. Der sich daraus ergebende Anfangsindex in der Speicherseite $F100 wird als PREFIX ($BF94) gespeichert.
Fügen wir diesem PREFIX einen weiteren Namen hinzu, z.B. „SUB2", wird Copy & Verify Pathname diesen Namen wieder auf $F100.. speichern, nämlich als (4) „SUB2" (0).
Das Anhängen dieses weiteren Namens an das PREFIX geschieht folgendermaßen (SET PREFIX, $D350):
- Vom Startindex des jetzigen PREFIX ($BF94) wird die Länge des neuen Namens abgezogen. Das funktioniert auch für den Fall, daß im Moment noch überhaupt kein PREFIX existiert, der „Startindex" ist in diesem Fall 00. Bei einer Kopie auf die Speicherseite $F100 ab dem Startindex ergibt sich damit automatisch das Namensende auf $F1FF.
- Im Kopier-Loop wird mit beiden Indexregistern gearbeitet, X enthält den Index zur Zieladresse, Y enthält den Index zur Quelle. Das Ende des Loops ist erreicht, wenn X wieder den Wert 00 hat, also wenn die Speicherstelle $F1FF ebenfalls beschrieben wurde. Da die Indexregister nur 8 Bit breit sind, ergibt sich für das dargestellte Beispiel der folgende Verlauf:
Start mit PREFIX von $BF94 (hier $ED) für die Quelle, Start des Ziel-Index mit $ED - 5 (= $E8). Der Speicherbereich $F1ED bis $F1FF wird auf $F1E8 bis $F1F9 kopiert. Danach hat das Y-Register (Quelle) den Wert 00, ist also übergelaufen, das X-Register hat den Wert $FA. Da das Loop-Ende durch das X-Register bestimmt wird, geht der Loop einfach weiter, aber mit $F100,00 (also dem neuen Namen) und endet mit der Kopie von $F100,5 auf $F100,FF.

**2) FIND FILE ENTRY mit gesetztem PREFIX**

Wird im Parameterblock des „user"-Programms ein vollständiger Name angegeben, (Beginn mit „/"), setzt Copy & Verify Pathname das FULL NAME Flag, der Startindex für den Vergleich „Name gefunden – Name gesucht" beginnt damit mit 00. Bei einem unvollständigen Namen wird dagegen immer PREFIX ($BF94) als Startindex für den ersten Namen im Path geladen.
Bei SEARCH VOLUME/PREFIX DIRECTORY ($D988) wird der VCB nach diesem ersten Namen des Path abgesucht. Wird der Name im VCB gefunden, hat das zwei Folgen:
– es wird nicht der Key Block der Volume Directory (00 02), sondern die Blocknummer des PREFIX Directory Key Blocks gesetzt und gelesen – dieser Key Block ist ebenfalls der VolDir Key Block, wenn das PREFIX nur aus einem Namen besteht;

– der erste Name im Path, also der Volume-Name, wird auf dem Volume selber nicht geprüft, falls eine Subdirectory PREFIXed ist.
Der nächste Schritt in SEARCH VOLUME ist der Vergleich „Name im Path – Name der geladenen Directory".
Für ein PREFIX, das nur aus einem Namen besteht, ändert das natürlich überhaupt nichts im Ablauf: Die Volume Directory ist die PREFIX-Directory, sie wird geladen und mit dem ersten Namen im (Gesamt)Path verglichen.
Für ein PREFIX mit zwei Namen (Volume/Subdirectory) ist der Volume-Name nach dem Auffinden im VCB uninteressant, hier muß der Startindex für den Vergleich „Directory-Name – Name gewünscht" auf den letzten Namen des PREFIX-Path gesetzt sein. Genauso verhält es sich für den Fall eines PREFIX mit mehr als zwei Namen, z.B. Volume/Sub1/Sub2/Sub3/PREFIXDIR. Hier wird die Überprüfung der Namen zwischen „Volume" und „PREFIXDIR" schlicht übersprungen(!).
Der entsprechende Loop befindet sich auf $D9DA. Hier wird mit dem von PREFIX kopierten Startindex innerhalb des „PREFIX-Path" begonnen. Die Längen der einzelnen Namen werden solange aufaddiert und das letzte Ergebnis zwischengespeichert, bis der Wert 00 erreicht ist, d.h. das Ende des „PREFIX-Path". Das (vor)letzte Ergebnis der Addition ergibt dann den Startindex des letzten Namens im „PREFIX-Path" und dieser letzte Name wird dann mit der geladenen Directory verglichen.
Was den Loop programmiertechnisch so interessant macht, ist die Tatsache, daß dieses Verfahren eben für ein PREFIX mit einer beliebigen Anzahl von Namen darin funktioniert.

# TEIL II – BOOT UND REBOOT VON PRODOS

## 2.1. Beschreibung in logischen Blöcken

### 2.1.1. Das RESET-Programm im Monitor

Unter der Voraussetzung, daß die Language Card nicht eingeschaltet ist und damit den F8-Rom überdeckt, ist der Ablauf (beim „Autostart"-ROM) vorgegeben, wenn der Prozessor des APPLE ein RESET-Signal erhält: Es erfolgt ein Sprung zu $FA62.
Nach diversen Initialisierungen (TEXT, NORMAL, PR#0 und IN#0 sind die wichtigsten) werden hier die Speicherstellen $03F3 und $03F4 miteinander verknüpft. Sind diese beiden Speicherstellen entsprechend gesetzt, wird zu der Adresse gesprungen, auf die $03F2/$03F3 zeigt (Warmstart).

**Die entsprechenden Startadressen:**
APPLESOFT ohne Disk-Betrieb: $E003
DOS 3.3 (48k): $9DBF
ProDOS: $FF69 beim Boot, danach $FF59, also in den Monitor (!)
BASIC.SYSTEM unter ProDOS: $BE00

Entspricht das Ergebnis der logischen Verknüpfung dagegen nicht den Erwartungen, nimmt das Programm an, daß der Computer gerade zum ersten Mal eingeschaltet wurde.
HOME wird ausgeführt und „APPLEII" auf den Schirm gebracht ($FAA6).
Die Slots werden der Reihe nach abgesucht, angefangen bei Slot 7. Enthält ein Slot einen ROM, dessen Bytes $01/$03/$05/$07 die Werte $20/$00/$03/$3C (DISKID) haben, springt das Programm zum Byte $00 des Slots, d.h. die Ausführung wird auf $Cx00 fortgesetzt (x = Nummer des gefundenen Slots). Ansonsten wird der nächstniedrigere Slot auf dieselbe Weise geprüft.
Wird bei dieser Suche auch in Slot 1 kein Controller identifiziert, erfolgt ein Kaltstart von APPLESOFT ($E000).

Anmerkung: Auch wenn ProDOS (im Gegensatz zu CP/M und UCSD-Pascal) den Controller in jedem Slot (außer 0) akzeptiert, wird für die folgende Besprechung angenommen, daß sich der Controller in Slot 6 befindet („Standard").

## 2.1.2. Das Boot-Programm des Controllers

| | |
|---|---|
| $C600-$C65B | Die ersten beiden Instruktionen sind von der Logik her sinnlos, aus ihnen besteht die erste Hälfte des DISKID.<br>Eine Tabelle zur Übersetzung der Nibbles (READ) wird im Bereich von $036C-$03D5 angelegt. Durch Aufruf eines „bekannten" RTS im Monitor wird die Slotadresse ermittelt und der entsprechende Index gespeichert (Slotno * 16). Damit ist die Adressierung der „State Machine", also des eigentlichen Disk-Controllers, slotunabhängig. |
| $C62F | Drive 1, INPUT, READ werden gesetzt, der Motor wird angeschaltet und der Arm über (dezimal) 40 Tracks zurückgezogen. Damit befindet er sich mit Sicherheit über Track 0. |
| $C652 | Es wird gesetzt: Track gewünscht ($0041) = 0, Sektor gewünscht ($003D) = 0, Zieladresse ($0026/27) = $0800. |
| $C65C | CLC |
| $C65D-C682 | Start mit CLC: Address Field wird gesucht.<br>Start mit SEC: Data Field wird gesucht.<br>Wartet, bis $D5/$AA/$xx auf der Diskette vorbeikommt. Wurde als drittes Byte $96 (Address Field) gefunden, weiter mit $C683. Wurde dagegen $AD gelesen, wird zuerst getestet, ob bereits ein Data Field gewünscht wurde. Wenn ja, Sprung zu $C6A6, ansonsten Sprung zu $C65C (Data Field nicht gefunden, nächstes Address Field). |
| $C683-$C6A5 | Lesen und Auswertung eines Address Field. Die Tracknummer wird auf $0040 gespeichert und mit $0041 (Track gewünscht) verglichen, die gefundene Sektornummer wird nur mit $003D (Sektor gewünscht) verglichen. Die Volume-Nummer wird übersprungen, Checksum und Trailer werden ignoriert. Stimmen Track und Sektor, ist das Carry durch den Vergleich gesetzt. Sprung zu $C65D: Jetzt Suche nach einem Data Field. Ansonsten Sprung zu $C65C. |
| $C6A6-$C6EA | Lesen eines Data Field. Die „zusätzlichen" Nibbles werden im Bereich $0300-$0355, die restlichen $100 Nibbles im Zielbereich gespeichert. |
| $C6CB | Lesen der Checksum. Wenn Fehler, zurück zu $C65C. |
| $C6D5 | Denibbling der gelesenen Daten in den Zielbereich ($0026/27) hinein. |
| $C6EB-C6FA | Die Zieladresse ($0026/27) wird um $100, die „Sektornummer gewünscht" ($003D) um eins erhöht und mit dem (ersten) geladenen Sektor, Byte $00 ($0800) verglichen. Ist die neue Sektornummer kleiner als der dort gespeicherte Wert, Sprung zum Lesen des nächsten Sektors, ansonsten Sprung zu $0801.<br>A enthält die Nummer des nächsten Sektors (2), X die Slotnummer * 16. |

## 2.1.3. Der Urlader von ProDOS

Der erste Teil des Urladers befindet sich auf Track 0, Sektor 0 und ist soeben geladen worden, der zweite Teil befindet sich auf Track 0, Sektor 2. Zusammen ergibt das den Block 00 00 auf der Diskette.
Der Urlader für SOS befindet sich auf Track 0, Sektoren $04 und $06, also auf Block 00 01 der Diskette.
Anmerkung: Diese Angaben beziehen sich auf die physikalischen Sektornummern, d.h. ohne vorherige Umsetzung („Sector skew").

| | |
|---|---|
| $0800 | Diese Speicherstelle hat den Wert $01: das Controller-Programm soll nur den Sektor 0 laden. |
| $0801 | Ein Vektor für APPLE III (?) wird übersprungen, die Nummer des nächsten Sektors mit dem Wert $03 verglichen. Aus der Slotno * 16 ergibt sich die Slotadresse der Controller-Karte. Hat das Byte $FF auf der Karte nicht den Wert $00, handelt es sich nicht um eine DiskII: Sprung zu $085B. |
| $0821 | War die Nummer des nächsten Sektors größer/gleich 3, ist dies bereits der zweite Durchgang. Sprung zu $0831. |
| $0823-$0830 | Die neue Obergrenze für das Controller-Programm wird auf $03 gesetzt, die „Sektornummer gewünscht" ($003D) auf $02. Durch Pushen der Slotadresse und des bekannten Einsprungs in die READ-Routine des Controllers mit anschließendem RTS wird Sektor $02 auf $0900-$09FF gelesen. Das Controller-Programm springt danach wieder zu $0801. |
| $0831-$0854 | Die READ-Routine des Controllers wird nach $09F7-$0A7E kopiert. Einige relative Sprünge werden verändert und einige Instruktionen angehängt. Die Routine endet hier nicht mehr mit einem Sprung zu $0801, sondern mit RTS. Der Einsprung liegt auf $0986. |
| $0855 | $0048/49 werden auf diesen Einsprung gesetzt. |
| $085B | Sprung hierher von $081F, wenn das Byte $CxFF des Controllers nicht den Wert 0 hat. Folge: $0048/49 zeigen auf Basisadresse des Controllers ($Cx) + Wert von $CxFF. |
| $0863-$0877 | Folgende Werte und Pointer auf der Zeropage werden gesetzt:<br>$0042: 01 (READ: nur für On-Card RWTBs anderer Controller);<br>$0046/47: 00 02 (Blocknummer);<br>$004A/4B und $0060/61: $0C00 (Zieladresse). |
| $0879-$0889 | Die Blocks 00 02 bis 00 05 (Volume Directory) werden via Aufruf von $0912 (GO READ) in den Bereich $0C00-$13FF geladen. |
| $088A | Die „Backward Reference" des ersten Blocks der geladenen Volume Directory muß 00 00 sein, ansonsten Sprung zu $08FF (ERROR). |

| | |
|---|---|
| $0896-08BD | Die geladene Directory wird nach dem Eintrag „PRODOS", Namenslänge 6, Storage Type 2 abgesucht. Wird dieser Eintrag nicht gefunden, Sprung zu $08FF (ERROR). |
| $08BE | Der erneute Test auf Storage Type 2 ist überflüssig. |
| $08C4 | Der gefundene File muß den Filetyp $FF (SYS) haben, ansonsten Sprung zu $08FF (ERROR). |
| $08CD | Bei Storage Type 2 enthält „erster Fileblock" den Indexblock, $0046/47 (Blocknummer) werden entsprechend gesetzt. |
| $08E3-08FB | Der nächste Block wird geladen. Beim ersten Mal ist das der Indexblock selber (auf $1E00), beim Loop sind es dann die Datenblocks, auf die der Indexblock zeigt. Sie werden ab $2000 geladen. |
| $08F8 | Wenn der Indexblock als nächstes auf einen Datenblock mit der Nummer 00 00 zeigt, ist der File zu Ende. Das entspricht **nicht** der Definition, ist aber hier ausreichend. |
| $08FC | **Sprung zu $2000: Start des PRODOS-Relocators** |
| $08FF | „Verlängerung" für Fehler: Sprung zu „UNABLE TO LOAD PRODOS" |
| $0902-$0911 | „PRODOS" in ASCII, mit Typ & Namenslänge für die Suchroutine in der Volume Directory. |
| $0912-$091A | „GO READ". $0044/45 werden auf die READ-Zieladresse gesetzt und die READ-Routine wird via $0048/49 angesprungen. $0048/49 sind entweder auf $0986 (DISKII) oder auf die RWTB auf der Controller-Karte (andere Drives) gesetzt. |
| $091D-$0923 | „Offsets" zur Basisadresse $09F2 zum Patchen der relativen Sprünge innerhalb der vom Controller kopierten READ-Routine. |
| $0924-$092A | Patch-Werte für die relativen Sprünge. |
| $092B-$092E | Neues Ende für die kopierte READ-Routine: RTS anstatt JMP $0801. |
| $092F | „Verlängerung" zum Arm Mover, ebenfalls an READ angehängt. |
| $0932-$093E | Diese Routine wird ebenfalls innerhalb des ProDOS-Urladers von nirgendwoher angesprungen (s. „Spezielles"). |
| $093F-$094F | ERROR: HOME und Ausgabe von: |
| $0950-$096C | „*** UNABLE TO LOAD PRODOS ***" in ASCII. |
| $096D-$0985 | Arm mover. Je nachdem, ob $0053 (momentane Arm-Position) erhöht oder erniedrigt wurde, wird die entsprechende Phase des Step-Motors geschaltet. |
| $0978-$0985 | Arm mover delay. Arbeitet mit konstanter Verzögerung. |

$0986-$09BB **READ BLOCK**
Aus der Blocknummer werden Track und Sektor berechnet. Diese Berechnung funktioniert nur für Blocknummern < $200. $003D erhält die berechnete Sektornummer, $0041 die gewünschte Tracknummer. $0045/46 (RWTB-Parameter: Zieladresse) werden auf $0026/27 (Zieladresse für die Routine von der Controller-Karte) kopiert.

$09AA Es werden zwei Sektoren (= 1 Block) gelesen. Wenn bei einem Fehler der Retry-Zähler abgelaufen ist, folgt RTS mit Carry gesetzt und damit „*** UNABLE TO LOAD PRODOS ***".

$09BC-$09E1 Steuer-Routine für den Arm mover. Die Positionierung erfolgt absolut, d.h. es wird dabei nichts von der Diskette gelesen. Diese Routine ist eine abgemagerte Version des späteren SEEKABS in der DISKII-RWTB.

$09CC Der gewünschte Track sollte erreicht sein. Sprung zu READ.

$09E2-$0A85 **READ SECTOR**
Es wird eine Maximalzahl von 127 Retries gesetzt. Die folgenden Routinen sind (bis auf veränderte relative Sprünge) eine exakte Kopie der READ-Routine vom Controller.

$09E8 Start des Loops: die Retries werden um eins heruntergezählt. Erreicht der Retry-Zähler den Wert 0, RTS mit gesetztem Carry.

$09EF Wartet auf einen Header. Wenn Address Field gefunden, Sprung zu $0A17. Wenn Data Field gefunden, Test auf „Data Field gewünscht" (SEC). Wenn ja, und drittes Byte = $AD, Sprung zu $0A3A. Ansonsten Sprung zu $09E8: Herunterzählen der Retries und nächster Versuch.

$0A17 Lesen und Auswertung eines Address Field. Wenn gewünschter Track/Sektor gefunden, folgt Rücksprung zu $09EE mit „Data Field gewünscht".
Stimmt die Sektornummer nicht, Sprung zu $09E8.
Stimmt die Tracknummer nicht, Sprung zu $09BC: der gelesene Track wird als „Track jetzt" gesetzt und der Arm erneut positioniert.

$0A3A Lesen eines Data Field. Stimmt die Checksum nicht, Rücksprung zu $09E8. Ansonsten Denibbling in den Zielbereich ($0026/27) und RTS zu READ BLOCK.

$0A83 „Verlängerung" für den Fall, daß die Tracknummer nicht stimmt (Read Address Field): Sprung zur erneuten Positionierung des Arms.

$0A85-$0BFF Von ProDOS beim Boot nicht belegt, d.h. dieser Speicherbereich bleibt beim Anschalten des APPLE vorläufig undefiniert („Müll").

$0C00-$13FF Vom Urlader geladene Blocks 00 02 bis 00 05 der Diskette: Volume Directory.

$1400-$1DFF Von ProDOS beim Boot nicht belegt („Müll").

$1E00-$1FFF Vom Urlader geladener Indexblock des Files PRODOS.

## 2.1.4. Der Relocator von ProDOS

Der File PRODOS wurde vom Urlader oder vom Kommando „-PRODOS" in den Bereich $2000-$5CFF geladen. Die einzige Speicherstelle, deren Wert übernommen wird, ist $0043 (Slotno * 16). Da beim Boot immer von Drive 1 geladen wird, entspricht der dort enthaltene Wert der Unitnummer (was bei Drive 2 nicht der Fall wäre).
Da das Programm PRODOS auch mit „-PRODOS" gestartet werden kann, ist es hier nicht möglich, auf die vom Urlader bereits geladene Volume Directory oder irgendwelche anderen Daten zurückzugreifen. Für die Suche nach dem ersten „.SYSTEM"-Programm muß die Directory deshalb erneut geladen und abgesucht werden. Im Fall eines „-PRODOS"-Kommandos stimmt die Unitnummer auch für Drive 2 eines Slots.

| | |
|---|---|
| $2000-$20D3 | **Hauptprogramm des Relocators** |
| $2000 | Die momentane Unitnummer wird in den Parameterblock eines späteren ONLINE gespeichert. |
| $2005 | TEXT/HOME und „PRODOS 1.0.1" etc. ($2448). |
| $2008 | Der erste Aufruf des MOVER ($2663) ist für jede Maschinenkonfiguration gleich:<br>Der Loader für das „.SYSTEM"-Programm wird nach $0800, die BRK/RESET-Vektoren nach $03F0 kopiert. Des weiteren wird die Prüfroutine für „128k" (RAMdisk) auf $0080 kopiert und die Speicherstellen $000A/0B erhalten den Wert $A.<br>Der darauffolgende Vektor zu ERR ($212A) ist sinnlos, es handelt sich hier um eine reine Kopieraktion ohne Fehlerprüfung. |
| $2016 | Test, ob der Computer mindestens 48k RAM hat. Wenn nicht, Sprung zu ERR ($212A). |
| $202E | Es wird anhand der Speicherstelle $FBB3 festgestellt, um welche Maschine (II/II+/IIe/IIc/III) es sich handelt und $0C (MACHID) entsprechend gesetzt ($23B0). Hier findet der erste Test auf II+-Raubkopien statt. |
| $2031 | Hat der Test bei $23B0 ergeben, daß nur 48k RAM vorliegen, werden die Startadressen der Parameterblocks für die folgenden Aufrufe des MOVER entsprechend versetzt und zeigen jetzt auf die Start- und Zieladressen für eine 48k-Maschine. Zusätzlich wird die Startadresse der RWTB für die DiskII von $F800 auf $B800 herabgesetzt. |
| $2047 | Zweiter Aufruf des MOVER.<br>Das MLI wird relokalisiert und kopiert, die RWTB für die DiskII und der IRQ/RESET-Handler innerhalb der Language Card werden kopiert.<br>Für 48k entfällt der IRQ/RESET-Handler, und die RWTB muß ebenfalls relokalisiert werden; diese Routinen sind bereits für den Betrieb ab $F800 ausgelegt.<br>Je nach Speichergröße wird eine vorbereitete Global Page entweder von $4E00 (64k) oder $4F00 (48k) kopiert. |
| $2055 | Der zweite Teil von MACHID ($000C) wird durch Test der Speicherstelle $FBC0 im F8-Rom gesetzt. Das ist eine Erweiterungsmöglichkeit |

für zukünftige Maschinen – für die Modelle II/II+/IIe bleibt MACHID in diesem Test unverändert. (Der IIc weist auf $FBC0 ein $00 auf).

$207F Alle Slots werden nach „bekannten" Karten (Thunderclock/80-Z/Controller) abgesucht ($24E6). MACHID wird dabei entsprechend verändert, wenn Uhr oder 80-Zeichen-Karte gefunden wurden. Hierbei findet auch der Aufruf der Routine statt, die „APPLEII" im ROM testet. MACHID wird dabei von $0C auf $BF98 kopiert.

$208E Der dritte Aufruf von MOVER: Je nach Speichergröße wird die Routine für die Thunderclock entweder nach $F142 oder $B142 verschoben und dabei mit der Slotnummer einer eventuell gefundenen Uhrenkarte ein zweites Mal modifiziert. Diese Verschiebung erfolgt immer, unabhängig davon, ob tatsächlich eine Uhrenkarte gefunden wurde.

$2099 Für eine 64k-Maschine erfolgt der vierte Aufruf des MOVER: REBOOT wird in die zweite Bank der Language Card kopiert. Sind nur 48k RAM vorhanden, wird dieser Aufruf übersprungen.

$20C8 Test von MACHID: Wurden zusätzliche 64k RAM gefunden, wird die RAMDisk initialisiert ($28FF). Diese Initialisierung erfolgt ohne vorherigen Test, d.h. das Kommando „-PRODOS" zerstört somit immer den Inhalt der RAMDisk.

$20D8 PRODOS KOMPLETT INSTALLIERT:
ONLINE wird mit dem Boot-Unit durchgeführt, der zurückgelieferte Volume-Name via SET PREFIX gesetzt und die Volume Directory via READ BLOCK ab $0C00 eingelesen.
Dieser Loop endet, wenn entweder eine Forward Reference mit 00 00 im zuletzt gelesenen Block gefunden wurde (Directory zu Ende) oder wenn vier Blocks gelesen wurden (nächste Zieladresse: $1400).

$2127 Sprung zu $0800, dem dorthin kopierten Loader für das nächste „.SYSTEM"-Programm.

Anmerkung: Der Loader für das „.SYSTEM"-Programm ist, um Verwirrungen zu vermeiden, auch im Listing auf seiner Originaladresse belassen worden. Die Besprechnung steht ab „$2234".

$212A-$2163 ERR – Nach jedem Aufruf des MOVER steht ein Vektor zu dieser Routine. Der MOVER endet nur mit Carry gesetzt, wenn beim Bestimmen der Befehlslängen zur Relokalisierung ein ungültiger Opcode gefunden wurde, d.h. bei reinen Kopieraktionen wird nie eine Fehlerbedingung entstehen.
ERR wird ebenfalls angesprungen, wenn weniger als 48k RAM festgestellt wurden.
Die Ausführung beschränkt sich auf HOME und das Anzeigen von „*** RELOCATION/CONFIGURATION ERROR ***", gefolgt von einer Endlosschleife auf $213B.

$213E „*** RELOCATION/CONFIGURATION ERROR ***" in ASCII.

| | |
|---|---|
| $2164-$2170 | Parameterblocks für die MLI-Aufrufe ONLINE, SET PREFIX und READ BLOCK nach erfolgter Installation. Die Unitnummer für ONLINE wird von $2002 gesetzt.<br>Die Zieladresse von READ BLOCK wird solange erhöht, bis die Volume Directory vollständig gelesen ist. |
| $2171-$2178 | Startadressen der Parameterblocks für MOVER (64k) |
| $2179-$217B | Startadressen der Parameterblocks für 48k, die Startadressen für den 2. und 3. Aufruf ($2173-76) werden damit ggf. überschrieben, der vierte Aufruf (Move von REBOOT) findet bei 48k nicht statt. |
| $2171-$2233 | Parameterblocks für den MOVER |

---

| | |
|---|---|
| $2234-$239F | FIND AND LOAD „.SYSTEM"-PROGRAM<br>Dieser Speicherbereich wird beim ersten Aufruf des MOVER auf $0800-$096C kopiert (nicht verschoben). |
| $2234-$2282 | Die gerade (neu) geladene Volume Directory wird nach einem Eintrag mit der Endung „.SYSTEM" abgesucht. |
| $224A ($0816) | Überschreitet der Eintrag, der als nächstes verglichen werden soll, den Bereich, in den die Volume Directory geladen wurde, folgt „UNABLE TO FIND A .SYSTEM FILE". |
| $2271 | „Verlängerung" für „UNABLE TO FIND A „.SYSTEM" FILE". |
| $2274 ($0840) | Jeder gefundene SYS-File, dessen Name länger als 8 Zeichen ist, wird vom Ende her mit „.SYSTEM" verglichen. Ist der Vergleich nicht erfolgreich, wird mit dem nächsten Eintrag weitergemacht. |
| $2283-$22E0 | File mit der Endung „.SYSTEM" und dem Typ $FF (SYS) gefunden. Der vollständige Name wird in den Parameterblock für den folgenden OPEN-Befehl kopiert, ein Leerzeichen angehängt. OPEN und GET EOF werden ausgeführt. |
| $22B0 ($087C) | Die von GET EOF zurückgelieferte Länge des Files wird getestet. Ist der File länger als $97FF Bytes, erfolgt die Meldung „SYSTEM PROGRAM TOO LARGE" und Abbruch. |
| $22C5 ($0891) | Der File wird via READ mit von SET EOF gesetztem Request count ab $2000 eingelesen. |
| $22D3 ($089F) | CLOSE (ALL) wird ausgeführt. |
| $22DE ($08AA) | **Sprung nach $2000, Start des „.SYSTEM"-Programms.** |
| $22E1-$2314 | Error Exits für den „.SYSTEM" Loader. Enden mit einer Endlosschleife auf $2313 ($08DF). |
| $22E1 ($08AD) | „UNABLE TO FIND A SYSTEM FILE"<br>Angesprungen, wenn die Directory erfolglos vollständig abgesucht wurde. |
| $22EE ($08BA) | „UNABLE TO LOAD .....SYSTEM"<br>Angesprungen, wenn bei OPEN, READ, GET EOF oder CLOSE ein Fehler auftrat. Der Name des Files wird eingesetzt. |

$2308 ($08D4) „SYSTEM PROGRAM TOO LARGE"
Angesprungen, wenn der File länger als $97FF Bytes ist oder wenn READ mit BAD BUFFER ADDRESS abbricht.

$2313 ($08DF) Endlosschleife. Alle drei Fehlermeldungen enden hier.

$2316-$2382 Fehlermeldungen in ASCII.

$2383 ($094F) Scratch für die Berechnung der Filenamen-Länge.

$2384-$2398 Parameterblocks für OPEN/GET EOF/READ/CLOSE des Loaders

$2399-$239F „.SYSTEM" in ASCII

**------- Ende des nach $0800 kopierten Bereichs ------**

$23A0-$23AF Vektoren für Page 3, werden beim ersten Aufruf des MOVER nach $03F0-$03FF kopiert.

$23B0-$2400 Erstes Setzen von MACHID, Test auf 64k und Test auf Raubkopien.

$23B9 Für einen APPLE II bleibt MACHID = $00.

$23C7 Wurde ein II+/III gefunden, wird mit der Speicherstelle $FB1E zusätzlich getestet und zwischen II+ und III (Emulation mode) unterschieden.

$23D4 Keine bekannte Maschine (d.h. Raubkopie): $0014/15 zeigen auf $0080, Y = $00, d.h. Byte $80 erhält den Wert $02 und beim darauffolgenden Sprung nach $0080 hängt sich der 6502 auf, weil das interne Mikroprogramm durcheinanderkommt.

$23DA Das Vorhandensein einer Language Card wird getestet und MACHID ($000C) entsprechend bereichert.

$23FE Sprung nach $0080 zum Test auf „128k"

$2401-$2447 Test auf „128k" und entsprechendes Setzen vonm MACHID ($000C) Diese Routine wird beim ersten Aufruf des MOVER nach $0080 kopiert.

$243C Von $000A/0B werden $500 mit Carry gelöscht abgezogen, dieser Pointer zeigt danach auf $FB09: „APPLE II" im F8-ROM. RTS zu $2031.

$2448-$2491 **TITLE PAGE**
Setzt TEXT, NORMAL, PR#0, IN#0 für den Fall, daß das BASIC.SYSTEM (oder ein anderes Programm) noch die CSW/KSW-Vektoren besetzt hält („-PRODOS"-Befehl) und und bringt „APPLE II" , „PRODOS 1.0.1" etc. auf den Schirm. COUT (Monitor: $FDED) wird dabei nicht benutzt.

$2492-$24E5 Text für TITLE PAGE in ASCII.

$24E6-$2626 **SEARCH SLOTS**
Die Slots werden der Reihe nach nach „bekannten" Karten abgesucht.

| | |
|---|---|
| $24F6 | Zuerst wird nach einem Controller gesucht: Die Bytes $Cx01/03/05 (nicht 07!, s. „Spezielles") werden mit DISKID verglichen. Handelt es sich nicht um einen Controller, Sprung zu $2569 mit Carry gesetzt. |
| $2507 | Das Byte $CxFF wird geladen. Hat es den Wert 0, wird der Controller als DiskII behandelt und der Einsprung in die karteneigene RWTB „getürkt", d.h. auf $F800 ($B800 für 48k RAM) gesetzt. Ansonsten wird als Start der RWTB die Kartenadresse + Wert von $CxFF angenommen. Aus $CxFE ergiben sich die 'Drive Characteristics', Bit 5 steht für „2 Drives möglich". Es wird ins Carry geschoben. |
| $252B | Der „Türke" für die DiskII: Anstelle der Slotadresse wird $F800 (B$800) untergeschoben. Das Carry wird gesetzt: 2 Laufwerke sind auf dem DiskII-Controller möglich. |
| $2535 | Das gefundene Unit (= Controller) wird in die DEVICE TABLE auf der Global Page eingetragen, je nach Status des Carry (1/2 Drives) wird nur der DRIVER START für Drive 1 ($BF11) oder zusätzlich der entsprechende Eintrag für Drive 2 ($BF21) gemacht. DEVNUM wird erhöht und das Carry gelöscht. |
| $2569 | Aufruf von $25B6: Einsprung mit Carry gesetzt: Suche nach anderen Karten in diesem Slot. Carry gelöscht: Controller gefunden, nur Eintrag in ROMBIT. |
| $256C | Herabsetzen der Slotnummer und Sprung zu $24F6, bis Slot 0 erreicht wird (Slot 0 wird nicht mehr getestet). |
| $2574 | Alle Slots erledigt: die Tabelle der Units in der Global Page wird so sortiert, daß die Einträge des Slots, von dem aus PRODOS gestartet wurde, als oberste in der Tabelle stehen. Damit wird die Suchreihenfolge von ProDOS (teilweise) festgelegt. |
| $25AA | „APPLE II" wird verifiziert ($2639). Fällt der Test positiv aus, kehrt die Routine mit dem Wert von $000C zurück, der jetzt in MACHID auf der Global Page ($BF98) gespeichert wird. RTS zu $2082 (oder $265C, s. „Spezielles").<br>Fällt der Test negativ aus, wird $00 zurückgebracht und es folgt ein Sprung zu $23D4. |
| $25B2-$2626 | Suche nach anderen „bekannten" Karten in diesem Slot. |
| $25B2 | Wenn in diesem Slot ein Controller gefunden wurde (Carry gelöscht), Sprung zu $2618. |
| $25B8 | Test auf Thunderclock-Karte. ID auf den Bytes $00/$02/$04/$06 mit den Werten $08/$28/$58/$70. Wird die Karte als Thunderclock identifiziert, wird MACHID bereichert und die Nummer des Slots wird in die Parameterblocks des MOVER eingetragen (48 und 64k). Sprung zu $2618.<br>Ist MACHID zu diesem Zeitpunkt noch $00 (eigentlich nicht möglich, s. „Spezielles"), folgt Sprung nach $25AA, d.h. kein Eintrag in MACHID! |
| $25DE | Test auf 80-Zeichen-Karte für IIe/c (nicht VIDEX!)<br>ID auf den Bytes $05/$07/$0B/$0C mit den Werten $38/$18/$01/$88 und |

entsprechendes Setzen von MACHID gefolgt von Sprung nach $2618, falls gefunden.
Falls MACHID = 0 ist, gilt dasselbe wie für die Uhrenkarte.

$2607 Kein „bekanntes" ROM in diesem Slot: Es wird festgestellt, ob die von dem Slot gelesene Information konstant ist (d.h. ob der Slot besetzt und „lesbar" ist). Diese Bedingung wird übrigens auch von einer RAM-Karte erfüllt. Falls ja, Sprung zu $2618, ansonsten RTS.

$2618 Eintrag des Slots in ROMBIT auf der Global Page und RTS zum Loop der Suchroutine.

$2627-$2628 Startadresse der RWTB für DiskII, wird für 48k auf $B800 gesetzt.

$2629-$262F DISKID und ID der Thunderclock ineinander verschachtelt

$2630 Das vierte Byte von DISKID. Nicht benutzt und wohl von einer früheren Version stehengeblieben.

$2631 Scratch für die Verify-Routine für „APPLEII"

$2632-$2638 Bit Mask für den ROMBIT-Eintrag.

$2639-2662 **Verify APPLEII**
Einsprung mit $000A/0B auf $FB09. Kleinschreibung wird in Großbuchstaben übersetzt. Return mit MACHID ($000C) in A bei positivem, $00 in A bei negativem Ausgang.

$2663 **MOVER**
Einsprung mit X/Y auf den Anfang des Parameterblocks gesetzt.

$2667 (Re)start der Hauptschleife: Der „Typ", die Zieladresse und die Anzahl werden gelesen. Ist der „Typ" größer als 5 => RTS

$2687 Ist der Typ = 0, Sprung zu $26EC: Löschen.
Ansonsten wird die Quelladresse gelesen und aus Zieladresse + Anzahl die Obergrenze berechnet.

$26A1 Ist der Typ = 1, Sprung zu $2710: Kopieren.
Für die Typen 2, 3 und 4 folgt jetzt im Parameterblock die Anzahl(!) der Referenz-Bereiche, für die die Relokalisierung „zugreifen" soll, danach die Unter- und Obergrenzen der Bereiche und schlußendlich der Faktor, um den die entsprechende Referenz erhöht (mit Overflow: erniedrigt) werden soll.

$26D4 Ist der Typ = 4, Sprung zu $26E6: Disassembly und Änderung. Sonst bleiben Typ 2 und 3 übrig: Änderung jedes (Typ 2) oder jedes zweiten Bytes (Typ 3). Typ 2 wird hier nicht benutzt.

$26EC Typ 0: Im Zielbereich werden „Länge" Bytes gelöscht, ausgehend von der im Parameterblock enthaltenen Startadresse. Vorher wird der Pointer im Parameterblock um die Anzahl der gelesenen Parameter erhöht ($2716). Sprung zu $2667.

$2710 Typ 1: Vor dem Sprung zur eigentlichen Copy-Routine wird der Pointer innerhalb des Parameterblocks erhöht ($2716).

$2716 Erhöhung des Pointers im Parameterblock um die Anzahl der gelesenen Bytes. RTS

$2723-2779 Damit es nicht zu übersichtlich wird, steht die Copy-Routine hier. Der „Handler" für Typ 1 springt zu $26DB. Von da her wird diese Routine aufgerufen, gefolgt von einem Sprung zu $2667 (nächster Parametersatz).

$2323 Vergleich von Ziel- und Quelladresse. Sind beide gleich, RTS.

$2734 Ist die Zieladresse < Quelladresse, wird in absteigender Reihenfolge kopiert, danach RTS.

$2757 Die Zieladresse ist größer als die Quelladresse: es wird in aufsteigener Reihenfolge kopiert, danach RTS.

$277A-$287B Typ 2/3: Innerhalb des Zielbereichs wird jedes (Typ 3: jedes zweite) Byte geprüft, ob es innerhalb der gesetzten Grenzwerte liegt und falls dem so ist, um den zugeordneten Wert erhöht ($27B8). Der Zielpointer wird solange um die Schrittweite erhöht ($27D4) und der Loop wiederholt, bis der obere Grenzwert erreicht ist. RTS zu $26DB, von wo aus die Kopierroutine aufgerufen wird. Diese Möglichkeit wird allerdings nicht genutzt: Quell- und Zieladressen sind bei allen hier verwendeten Parametersätzen gleich, d.h. die Kopierroutine kehrt sofort zur Hauptschleife des MOVER zurück.

$278C-$27B7 Typ 4: Innerhalb des Zielgebiets wird die Länge jeder Instruktion bestimmt ($27E7). Hier ist die einzige Möglichkeit für eine Fehlerbedingung innerhalb des MOVER.
Ist die Länge der Instruktion = 3 (absolute Referenz), wird $27B8 aufgerufen und korrigiert diese Referenz, falls sie in den Zielbereiche(n) liegt um den jeweils dem Zielbereich zugeordneten Wert.
Der Zielpointer wird um die Länge der Instruktion erhöht ($27D4). Die Schleife endet, wenn die Obergrenze erreicht ist. RTS zu $DB6B, von wo aus die Kopier-Routine aufgerufen wird. Diese Möglichkeit wird allerdings nicht genutzt (Quell- und Zielpointer sind in den gegebenen Parametern immer identisch).

$27A6 FEHLER: Die Längenbestimmung hat den Wert 0 zurückgeliefert (= ungültiger Opcode). Rückkehr mit gesetztem Carry zu der Routine, die den MOVER aufgerufen hat.
=> RELOCATION/CONFIGURATION ERROR.

$27B0 Ein RTS zum MOVER(?) mit gesetztem Carry, der von nirgendwo her angesprungen wird (s. „Spezielles").

$27B8 Diese Routine überprüft, ob das n.te Byte einer Instruktion (Referenz-Hibyte) innerhalb des Bereichs liegt und addiert (mit Overflow: subtrahiert!) den ebenfalls zugeordneten Wert. Dabei kann es mehrere Bereiche geben. RTS zur aufrufenden Routine.

$27D4-$27E6 Der Zielpointer wird um die Länge der Instruktion (für Typ 2/3: Schrittweite) erhöht und mit der Obergrenze verglichen, das Carry wird dabei entsprechend gesetzt. RTS zur aufrufenden Routine.

$27E7-$27FB Die Länge einer Instruktion wird bestimmt. Für ungültige Opcodes wird der Wert 0 gesetzt. RTS zur aufrufenden Routine (nur von Typ 4 benutzt).

$27FC-$283B Tabelle zur Bestimmung der Instruktionslängen

$283C Schrittweite für Typ 2/3

$283D Anzahl der Referenz-Bereiche (-1) für Typ 2/3/4.

$283E-$2855 Unter- und Obergrenzen (= Ref.-Bereich), Additionswerte

$2855-$28FF Unbenutzt (Nullen)

$28FF-293C **RAMDISK INSTALLATION**
Das RAMDisk-Interface wird von $2C00 auf $FF00-$FF99 kopiert, die RAMDisk-RWTB von $2000 auf $0200-$03FF nach AUX.
Auf der Global Page wird ein (weiteres) Unit eingetragen: Slot 3, Drive 2. DEVNUM wird um eins erhöht.
Diese Routine wird nach dem Absuchen der Slots und dem „Sortieren" der Einträge in die DRIVE TABLE ausgeführt. Die RAMDisk ist damit immer das oberste (erste) Unit in der DRIVE TABLE.

$293D-$29FF unbenutzt (Nullen)

$2A00-$2BFF RAMDisk-RWTB: nach AUX $0200-$03FF kopiert (nur 128k)

$2C00-$2C99 RAMDisk-Interface: auf $FF00 kopiert (nur 128k)

$2D00-$4DFF MLI: auf $D000 (48k: $9000) verschoben

$4E00-$4EFF Global Page für 64/128k

$4F00-$4FFF Global Page für 48k

$5000-$507D Thunderclock-Routine: auf $F142 (48k: $B142) verschoben

$5080-$50FF IRQ/RESET- Handler: auf $FF80 verschoben (nur 64/128k)

$5100-$51FF Unbenutzt (Nullen)

$5200-$58FF RWTB für DiskII: auf $F800 kopiert (48k: auf $B800 verschoben)

$5900-$61FF REBOOT: auf $D100 (Bank2) kopiert (nur 64/128k)

# 2.2. Kommentierte Listings des Boot/Relocators

## 2.2.1. Das RESET-Programm im „Autostart"-Monitor

```
FA62-  D8          CLD             ; unbedingt nötig, sonst Absturz!
FA63-  20 84 FE    JSR  $FE84      ; SETNORM (nicht INVERSE)
FA66-  20 2F FB    JSR  $FB2F      ; INIT - setzt TEXT, MIX, LORES
FA69-  20 93 FE    JSR  $FE93      ; SETKBD („IN#0")
FA6C-  20 89 FE    JSR  $FE89      ; SETVID („PR#0")
FA6F-  AD 58 C0    LDA  $C058      ; Announciator 0 auf HI
FA72-  AD 5A C0    LDA  $C05A      ; Announciator 1 auf HI
FA75-  AD 5D C0    LDA  $C05D      ; Announciator 2 auf LO
FA78-  AD 5F C0    LDA  $C05F      ; Announciator 3 auf LO
FA7B-  AD FF CF    LDA  $CFFF      ; CLRROM - Alle $C800-ZusatzROMs aus
FA7E-  2C 10 C0    BIT  $C010      ; KBDSTRB - löscht evtl. Tastaturzeichen
FA81-  D8          CLD
FA82-  20 3A FF    JSR  $FF3A      ; BELL
FA85-  AD F3 03    LDA  $03F3      ; RESET-Handler, Startadresse hi
FA88-  49 A5       EOR  #$A5
FA8A-  CD F4 03    CMP  $03F4      ; PWRUP-Byte
FA8D-  D0 17       BNE  $FAA6      ; Sprung für Kaltstart
FA8F-  AD F2 03    LDA  $03F2      ; RESET-Handler, Startadresse low
FA92-  D0 0F       BNE  $FAA3      ; wenn <> 0, dann Warmstart
FA94-  A9 E0       LDA  #$E0
FA96-  CD F3 03    CMP  $03F3      ; RESET-Adresse auf $E000 (Kaltstart BASIC)?
FA99-  D0 08       BNE  $FAA3      ; nein, Warmstart
FA9B-  A0 03       LDY  #$03       ; RESET ist auf BASIC-Kaltstart: nächstes
FA9D-  8C F2 03    STY  $03F2        Mal Start mit $E003 (BASIC-Warmstart)
FAA0-  4C 00 E0    JMP  $E000      ; BASIC - KALTSTART

FAA3-  6C F2 03    JMP  ($03F2)    ; Warmstart - Sprung zum Handler
```

– KALTSTART

```
FAA6-  20 60 FB    JSR  $FB60      ; HOME und Ausgabe von „APPLE II"
FAA9-  A2 05       LDX  #$05
FAAB-  BD FC FA    LDA  $FAFC,X    ; Setzen der Page 3-Vektoren
FAAE-  9D EF 03    STA  $03EF,X      für BRK etc, nicht RESET!
FAB1-  CA          DEX
FAB2-  D0 F7       BNE  $FAAB
FAB4-  A9 C8       LDA  #$C8
FAB6-  86 00       STX  $00        ; = 00
FAB8-  85 01       STA  $01        ; = $C8, erster Slot: $C700
FABA-  A0 07       LDY  #$07
FABC-  C6 01       DEC  $01
FABE-  A5 01       LDA  $01
FAC0-  C9 C0       CMP  #$C0       ; bereits bei Slot 0?
FAC2-  F0 D7       BEQ  $FA9B      ; ja, RESET auf BASIC, Kaltstart von BASIC
FAC4-  8D F8 07    STA  $07F8
FAC7-  B1 00       LDA  ($00),Y    ; $Cx07/05/03/01
FAC9-  D9 01 FB    CMP  $FB01,Y    ; DISKID: $20/**/$00/**/$03/**/$3C
FACC-  D0 EC       BNE  $FABA      ; kein Controller, nächster Slot
FACE-  88          DEY
FACF-  88          DEY
FAD0-  10 F5       BPL  $FAC7      ; Vergleich bis herunter zu Byte $Cx01
FAD2-  6C 00 00    JMP  ($0000)    ; ok. Sprung zu $Cx00
```

## 2.2.2. Das Boot-Programm auf der DISKII Controller-Karte (hier in Slot 6)

```
C600-  A2 20     LDX  #$20        ; DISKID: $20/$00/$03/$3C, das vierte Byte
C602-  A0 00     LDY  #$00          wird nur vom Monitor getestet!
C604-  A2 03     LDX  #$03        ; durch den rechnerischen Aufbau einer
C606-  86 3C     STX  $3C           READ TRANSLATION TABLE wird der Bereich
C608-  8A        TXA                $0356-$03FF belegt. Dabei enthält nur der
C609-  0A        ASL                Bereich $036C-$03D5 tatsächlich verwert-
C60A-  24 3C     BIT  $3C           bare Daten, der Rest ist „Abfall".
C60C-  F0 10     BEQ  $C61E
C60E-  05 3C     ORA  $3C         ; $036C: 00 01 ** ** 02 03 ** 04 05 ...
C610-  49 FF     EOR  #$FF          für    96 97 98 99 9A 9B 9C 9D 9E ...
C612-  29 7E     AND  #$7E
C614-  B0 08     BCS  $C61E         Übersetzung durch direkte Adressierung
C616-  4A        LSR                mit dem gelesenen „Disk"-Byte, die Werte
C617-  D0 FB     BNE  $C614         $98/$99/$9C/$A0... stellen ungültige
C619-  98        TYA                „Disk"-Bytes dar!
C61A-  9D 56 03  STA  $0356,X
C61D-  C8        INY
C61E-  E8        INX
C61F-  10 E5     BPL  $C606

C621-  20 58 FF  JSR  $FF58       ; „bekanntes" RTS im Monitor
C624-  BA        TSX
C625-  BD 00 01  LDA  $0100,X     ; das ist das Hibyte der Returnadresse: $Cx
C628-  0A        ASL
C629-  0A        ASL              ; das linke Nibble wird herausgeshiftet,
C62A-  0A        ASL                also $Cx => $x0
C62B-  0A        ASL
C62C-  85 2B     STA  $2B         ; = Slotnummer * 16
C62E-  AA        TAX
```

– Setzt Drive 1 auf READ

```
C62F-  BD 8E C0  LDA  $C08E,X     ; INPUT (d.h. READ)
C632-  BD 8C C0  LDA  $C08C,X     ; Strobe (Setzen des Befehls)
C635-  BD 8A C0  LDA  $C08A,X     ; SELECT DRIVE 1
C638-  BD 89 C0  LDA  $C089,X     ; MOTOR ON
```

– Zieht den Arm über 40 Tracks zurück

```
C63B-  A0 50     LDY  #$50        ; 80 (dezimal) halbe Tracks => 40 Tracks
C63D-  BD 80 C0  LDA  $C080,X     ; Basisadresse des Arm-Motors: PHASE OFF
C640-  98        TYA
C641-  29 03     AND  #$03        ; ergibt 03/02/01/00/03/02/01/00...
C643-  0A        ASL              ; * 2 = 06/04/02/01 => Phasen-Adressen
C644-  05 2B     ORA  $2B
C646-  AA        TAX
C647-  BD 81 C0  LDA  $C081,X     ; Basisadresse des Arm-Motors: PHASE ON
C64A-  A9 56     LDA  #$56
C64C-  20 A8 FC  JSR  $FCA8       ; WAIT, RTS mit A = 00
C64F-  88        DEY              ; Ab- und Anschalten der Phasen in
C650-  10 EB     BPL  $C63D         absteigender Richtung: STEP OUT

C652-  85 26     STA  $26         ; = 00: Zieladresse lo
C654-  85 3D     STA  $3D         ; = 00: Sektor gewünscht
C656-  85 41     STA  $41         ; = 00: Track gewünscht
C658-  A9 08     LDA  #$08
C65A-  85 27     STA  $27         ; Zieladresse hi => $0026/27 = $0800

C65C-  18        CLC              ; Indikator: Address Field gesucht

C65D-  08        PHP
```

```
C65E-  BD 8C C0    LDA   $C08C,X
C661-  10 FB       BPL   $C65E
C663-  49 D5       EOR   #$D5          ; $D5
C665-  D0 F7       BNE   $C65E
C667-  BD 8C C0    LDA   $C08C,X
C66A-  10 FB       BPL   $C667
C66C-  C9 AA       CMP   #$AA          ; $AA
C66E-  D0 F3       BNE   $C663
C670-  EA          NOP
C671-  BD 8C C0    LDA   $C08C,X
C674-  10 FB       BPL   $C671
C676-  C9 96       CMP   #$96
C678-  F0 09       BEQ   $C683         ; $96 => Address Field Header gefunden
C67A-  28          PLP
C67B-  90 DF       BCC   $C65C         ; kein Data Field gewünscht, Retry
C67D-  49 AD       EOR   #$AD          ; ok. Data Field gewünscht. Ist drittes
C67F-  F0 25       BEQ   $C6A6           Byte $AD?
C681-  D0 D9       BNE   $C65C         ; nein, nächstes Address Field
```

– Lesen und Auswertung eines Address Field

```
C683-  A0 03       LDY   #$03          ; 3 Bytes zu lesen: Checksum wird ignoriert
C685-  85 40       STA   $40           ; hier bleibt „Trackno gefunden" beim Exit
C687-  BD 8C C0    LDA   $C08C,X
C68A-  10 FB       BPL   $C687
C68C-  2A          ROL
C68D-  85 3C       STA   $3C           ; „erste Hälfte" eines even-odd Bytes
C68F-  BD 8C C0    LDA   $C08C,X
C692-  10 FB       BPL   $C68F
C694-  25 3C       AND   $3C           ; zweite Hälfte
C696-  88          DEY                 ; am Ende des Loop bleibt „Sektor gefunden"
C697-  D0 EC       BNE   $C685           im Akku
C699-  28          PLP                 ; „Address Field/Data Field"-Indikator
C69A-  C5 3D       CMP   $3D           ; Vergleich: „Sektorno gefunden-gewünscht"
C69C-  D0 BE       BNE   $C65C         ; ungleich: nächstes Address Field
C69E-  A5 40       LDA   $40
C6A0-  C5 41       CMP   $41           ; „Track gefunden - gewünscht"
C6A2-  D0 B8       BNE   $C65C         ; ungleich - nächstes Address Field
C6A4-  B0 B7       BCS   $C65D         ; gleich => C gesetzt: „Data Field gewünscht"
```

– Lesen eines Data Field

```
C6A6-  A0 56       LDY   #$56          ; $56 „zusätzliche" Bytes von der Disk
C6A8-  84 3C       STY   $3C             enthalten die LSBs der „normalen" Bytes
C6AA-  BC 8C C0    LDY   $C08C,X         und werden in absteigender Reihenfolge
C6AD-  10 FB       BPL   $C6AA           gelesen
C6AF-  59 D6 02    EOR   $02D6,Y       ; RDTRANS, Bereich $036C-$03D5
C6B2-  A4 3C       LDY   $3C
C6B4-  88          DEY
C6B5-  99 00 03    STA   $0300,Y       ; „zusätzliche" Bytes auf $0355-$0300
C6B8-  D0 EE       BNE   $C6A8         ; Z-Flag gesetzt von „DEY"
C6BA-  84 3C       STY   $3C           ; = 00 => $100 „normale" Bytes
C6BC-  BC 8C C0    LDY   $C08C,X
C6BF-  10 FB       BPL   $C6BC
C6C1-  59 D6 02    EOR   $02D6,Y       ; RDTRANS
C6C4-  A4 3C       LDY   $3C
C6C6-  91 26       STA   ($26),Y       ; direkt in den Zielbereich in
C6C8-  C8          INY                   aufsteigender Reihenfolge
C6C9-  D0 EF       BNE   $C6BA
C6CB-  BC 8C C0    LDY   $C08C,X       ; Checksum
C6CE-  10 FB       BPL   $C6CB
C6D0-  59 D6 02    EOR   $02D6,Y
C6D3-  D0 87       BNE   $C65C         ; Checksum-Fehler: nächstes Address Field
```

– Denibble der gelesenen Daten

```
C6D5-  A0 00      LDY   #$00
C6D7-  A2 56      LDX   #$56          ; $56 „zusätzliche" Bytes
C6D9-  CA         DEX                   im Bereich $0355 bis $0300 („abwärts"):
C6DA-  30 FB      BMI   $C6D7
C6DC-  B1 26      LDA   ($26),Y       ; die beiden LSBs werden wieder in die
C6DE-  5E 00 03   LSR   $0300,X         „normalen" Bytes hineingeschoben
C6E1-  2A         ROL
C6E2-  5E 00 03   LSR   $0300,X
C6E5-  2A         ROL                 ; zweites „zusätzliches" Bit
C6E6-  91 26      STA   ($26),Y
C6E8-  C8         INY                 ; Loop-Ende bestimmt durch Y = 00, d.h.
C6E9-  D0 EE      BNE   $C6D9           X-Reg wird dreimal mit $56 geladen
                                        (erstes/zweites/drittes Drittel)
C6EB-  E6 27      INC   $27           ; Zieladresse + $100
C6ED-  E6 3D      INC   $3D           ; „Sektornummer gewünscht" + 1
C6EF-  A5 3D      LDA   $3D
C6F1-  CD 00 08   CMP   $0800         ; weitere Sektoren zu lesen?
C6F4-  A6 2B      LDX   $2B           ; Slotno X* 16
C6F6-  90 DB      BCC   $C6D3         ; ja, nächster Sektor
C6F8-  4C 01 08   JMP   $0801         ; nein, Boot Stufe 2

C6FB-  00         ; „unbenutzt", für ProDOS gilt:
C6FC-  00 00      ; TOTAL BLOCKS: Gesamtkapazität (lo-hi)
C6FE-  00         ; DRIVE CHARACTERISTICS
C6FF-  00         ; Startadresse der On-Card RWTB (low)
```

## 2.2.3. Der ProDOS-Urlader (Track 0 – Sektoren 00 und 02, d.h. Block 00 00)

```
0800-  01                           ; erster Einsprung: nur Sektor 0 geladen
0801-  38         SEC
0802-  B0 03      BCS   $0807       ; „always"
0804-  4C 32 A1   JMP   $A132       ; APPLE III?
0807-  86 43      STX   $43         ; Slotno * 16
0809-  C9 03      CMP   #$03        ; $3B: nächste Sektorno. War das der
080B-  08       * PHP                 erste Einsprung?
080C-  8A         TXA               ; Slotno*16
080D-  29 70      AND   #$70
080F-  4A         LSR
0810-  4A         LSR
0811-  4A         LSR
0812-  4A         LSR               ; /16 = Slotno
0813-  09 C0      ORA   #$C0
0815-  85 49      STA   $49         ; $C6 für Slot 6
0817-  A0 FF      LDY   #$FF
0819-  84 48      STY   $48
081B-  28       * PLP               ; vom Vergleich „Sektornummer = 3"
081C-  C8         INY
081D-  B1 48      LDA   ($48),Y     ; ($48) => $CxFF
081F-  D0 3A      BNE   $085B       ; Sprung, wenn keine DiskII
0821-  B0 0E      BCS   $0831       ; vom „PLP": C gesetzt bei 2. Einsprung
0823-  A9 03      LDA   #$03        ; erster Entry: Setzen von max-sector auf 2
0825-  8D 00 08   STA   $0800
0828-  E6 3D      INC   $3D         ; Überspringen des (physikalischen) Sektor 1
082A-  A5 49      LDA   $49         ; $Cx ($C0 + Slotnummer)
082C-  48         PHA
082D-  A9 5B      LDA   #$5B        ; Startadresse von „READ Sektor" auf der
082F-  48         PHA                 Controller Card: $Cx5C
0830-  60         RTS               ; zur Contr. Card, danach wieder zu $0801
```

```
0831-  85 40      STA   $40          ; zweiter Entry: A = 3
0833-  85 48      STA   $48
0835-  A0 63      LDY   #$63
0837-  B1 48      LDA   ($48),Y      ; $Cx03
0839-  99 94 09   STA   $0994,Y
083C-  C8         INY                ; $Cx66-ED werden nach $09F7-$0A7E kopiert
083D-  C0 EB      CPY   #$EB           (die „READ Sektor"-Routine von der Card)
083F-  D0 F6      BNE   $0837

0841-  A2 06      LDX   #$06         ; 7 relative Sprünge in dem Block zu ändern
0843-  BC 1D 09   LDY   $091D,X      ; Offsets zur Basis-Patchadresse $09F2
0846-  BD 24 09   LDA   $0924,X      ; neue Sprungweiten
0849-  99 F2 09   STA   $09F2,Y
084C-  BD 2B 09   LDA   $092B,X      ; 7 Befehls-Bytes werden angehängt: RTS
084F-  9D 7F 0A   STA   $0A7F,X        anstelle des Exits zu $0801
0852-  CA         DEX
0853-  10 EE      BPL   $0843

0855-  A9 09      LDA   #$09         ; Pointer zur kopierten READ-Routine:
0857-  85 49      STA   $49            $0048/49 zeigen auf $0986
0859-  A9 86      LDA   #$86
```

Xref: $081F – $CxFF auf der Karte (nicht DISKII) enthält das Lobyte der Startadresse der On-Card RWTB.

```
085B-  A0 00      LDY   #$00
085D-  C9 F9      CMP   #$F9         ; Lobyte > $F8?
085F-  B0 2F      BCS   $0890        ; ERROR (keine READ-Routine auf der Card)
0861-  85 48      STA   $48          ; = $86 (DISKII) oder Lobyte der RWTB
0863-  84 60      STY   $60          ; = $00, Zieladresse lo
0865-  84 4A      STY   $4A          ; Pointer lo  für Directory-Absuchen/Index
0867-  84 4C      STY   $4C          ; Pointer lo für Indexblock (obere Hälfte)
0869-  84 4E      STY   $4E          ; Index innerhalb des Indexblocks
086B-  84 47      STY   $47          ; Blocknummmer hi
086D-  C8         INY
086E-  84 42      STY   $42          ; = 1: READ – nur für On-Card RWTB
0870-  C8         INY
0871-  84 46      STY   $46          ; nächster Block zu lesen: Nummer 00 02
0873-  A9 0C      LDA   #$0C         ; Zieladresse $0C00
0875-  85 61      STA   $61
0877-  85 4B      STA   $4B          ; Pointer hi für Directory-Absuche
```

**– LOAD DIRECTORY** (Zielbereich $0C00 bis $13FF)

```
0879-  20 12 09   JSR   $0912        ; READ BLOCK (2 Sektoren)
087C-  B0 68      BCS   $08E6        ; ERROR
087E-  E6 61      INC   $61
0880-  E6 61      INC   $61          ; Zieladresse + $200
0882-  E6 46      INC   $46          ; Blocknummer lo (hi = 00)
0884-  A5 46      LDA   $46
0886-  C9 06      CMP   #$06         ; Block 6 erreicht?
0888-  90 EF      BCC   $0879        ; Loop für Blocks 2/3/4/5 (Volume Directory)

088A-  AD 00 0C   LDA   $0C00        ; Backward Reference des Key Blocks: muß
088D-  0D 01 0C   ORA   $0C01          00 00 sein, sonst
0890-  D0 6D      BNE   $08FF        ; ERROR (auch von $085B:$CxFF<>0 und >$F8)
0892-  A9 04      LDA   #$04         ; Basis des ersten Eintrags: $C04,
0894-  D0 02      BNE   $0898          der Key-Eintrag muß übersprungen werden

0896-  A5 4A      LDA   $4A          ; $004A/4B auf nächsten Eintrag (NEXT)
0898-  18         CLC
0899-  6D 23 0C   ADC   $0C23        ; + Länge eines Eintrags (Byte $23) im Key
```

```
089C-  A8          TAY
089D-  90 0D       BCC   $08AC
089F-  E6 4B       INC   $4B          ; Übertrag auf Hibyte des Pointers
08A1-  A5 4B       LDA   $4B
08A3-  4A          LSR
08A4-  B0 06       BCS   $08AC        ; Hibyte ungerade, => innerhalb eines Blocks
08A6-  C9 0A       CMP   #$0A         ; nächster DIR-Block: $13FF überschritten?
08A8-  F0 55       BEQ   $08FF        ; Ja, FILE NOT FOUND => ERROR
08AA-  A0 04       LDY   #$04         ; Nein, Start im nächsten DIR-Block

08AC-  84 4A       STY   $4A
08AE-  AD 02 09    LDA   $0902
08B1-  29 0F       AND   #$0F         ; $26 => Namenslänge = 6
08B3-  A8          TAY

08B4-  B1 4A       LDA   ($4A),Y      ; Vergleich e. Eintrags m. intern
08B6-  D9 02 09    CMP   $0902,Y        gespeicherten „PRODOS"
08B9-  D0 DB       BNE   $0896        ; Namen sind nicht gleich
08BB-  88          DEY
08BC-  10 F6       BPL   $08B4        ; Loop bis Namensanfang, incl. Type/Länge
08BE-  29 F0       AND   #$F0         ; Namen sind gleich, der nochmalige
08C0-  C9 20       CMP   #$20         ; Vergleich des Storage Type ist unnötig
08C2-  D0 3B       BNE   $08FF        ; ERROR
08C4-  A0 10       LDY   #$10
08C6-  B1 4A       LDA   ($4A),Y      ; Byte $10 im File-Eintrag: File Type
08C8-  B9 FF       CMP   #$FF         ; muß .SYS sein (Type $FF)
08CA-  D0 33       BNE   $08FF        ; ERROR
08CC-  C8          INY
08CD-  B1 4A       LDA   ($4A),Y      ; Bytes $11/$12 im File-Eintrag: „erster
08CF-  85 46       STA   $46            Block im File", bei Storage Type 2 ist
08D1-  C8          INY                  das der einzige Indexblock
08D2-  B1 4A       LDA   ($4A),Y
08D4-  85 47       STA   $47          ; Nummer des Indexblocks - hi

08D6-  A9 00       LDA   #$00
08D8-  85 4A       STA   $4A          ; $004A/4B: Pointer zur unteren Hälfte
08DA-  A0 1E       LDY   #$1E           des Indexblocks: $1E00
08DC-  84 4B       STY   $4B            $004C/4D: Pointer zur oberen Hälfte
08DE-  84 61       STY   $61            des Indexblocks: $1F00
08E0-  C8          INY                  $0060/61: Zieladresse für READ des
08E1-  84 4D       STY   $4D            Indexblocks: $1E00

08E3-  20 12 09    JSR   $0912        ; BLOCKREAD, beim 1. Mal den Indexblock
08E6-  B0 17       BCS   $08FF          selber, danach die Blocks, auf die der
08E8-  E6 61       INC   $61            Indexblock zeigt (Zieladresse: ab $2000)
08EA-  E6 61       INC   $61          ; Zieladresse + $200
08EC-  A4 4E       LDY   $4E          ; Index innerhalb des Blocks, Start mit 00
08EE-  E6 4E       INC   $4E
08F0-  B1 4A       LDA   ($4A),Y      ; untere Hälfte des Indexblocks
08F2-  85 46       STA   $46          ; = Blocknummer lo des nächsten Datenblocks
08F4-  B1 4C       LDA   ($4C),Y      ; obere Hälfte des Indexblocks
08F6-  85 47       STA   $47          ; = Blocknummer hi des nächsten Datenblocks
08F8-  11 4A       ORA   ($4A),Y
08FA-  D0 E7       BNE   $08E3        ; Loop bis zu einem Block mit Nummer 00 00

08FC-  4C 00 20    JMP   $2000        ; START PRODOS RELOCATOR

08FF-  4C 3F 09    JMP   $093F        ; Error Exit: „UNABLE TO LOAD"

0902- 26 50 52 4F 44 4F 53 20  20 20 20 20 20 20 20 20    &PRODOS
```

**– GO READ**

```
0912-   A5 60       LDA   $60          ; Zieladresse für READ
0914-   85 44       STA   $44          ; => RWTB-Parameter
0916-   A5 61       LDA   $61
0918-   85 45       STA   $45          ; Hibytes
091A-   6C 48 00    JMP   ($0048)      ; $0986 für DISKII, ansonsten On-Card RWTB

091D- 08 1E 24 3F 45 47 76             ; Patches f.relative Sprünge: Offset zu $09F2
0924- F4 D7 D1 B6 4B B4 AC             ; neue Sprungweiten
```

– an READ ab $0A7F angehängt (7 Bytes)

```
092B-   A6 2B       LDX   $2B          ; Slotno * 16
092D-   18          CLC                ; Read o.k.
092E-   60          RTS                ; anstelle von „JMP $0801"
092F-   4C BC 09    JMP   $09BC        ; „Verlängerung" zum Retry
```

– ?????

```
0932-   A9 9F       LDA   #$9F
0934-   48          PHA                ; pusht „Returnadresse" $A000
0935-   A9 FF       LDA   #$FF
0937-   48          PHA                (s. „Spezielles")
0938-   A9 01       LDA   #$01
093A-   A2 00       LDX   #$00
093C-   4C 79 F4    JMP   $F479        ; Apple III / SOS?
```

**– ERROR**

```
093F-   20 58 FC    JSR   $FC58        ; HOME (Monitor)
0942-   A0 1C       LDY   #$1C
0944-   B9 50 09    LDA   $0950,Y      ; „*** UNABLE TO LOAD PRODOS ***"
0947-   99 AE 05    STA   $05AE,Y      ; wird direkt in den Bildspeicher kopiert
094A-   88          DEY
094B-   10 F7       BPL   $0944
094D-   4C 4D 09    JMP   $094D        ; Loop bis RESET („Aufgehängt")

0950- AA AA AA A0 D5 CE C1 C2  CC C5 A0 D4 CF A0 CC CF    *** UNABLE TO LO
0960- C1 C4 A0 D0 D2 CF C4 CF  D3 A0 AA AA AA             AD PRODOS ***
```

**– ARM MOVER**

```
096D-   A5 53       LDA   $53          ; Entry1: „new track" (INC'd/DEC'd - ON)
096F-   29 03       AND   #$03         ; Entry2: „old track" (PHASE OFF)
0971-   2A          ROL                ; wenn Carry set: PHASE ON, sonst OFF
0972-   05 2B       ORA   $2B          ; Slotno * 16
0974-   AA          TAX
0975-   BD 80 C0    LDA   $C080,X      ; Phasenadressierung des Stepper-Motors

0978-   A9 2C       LDA   #$2C         ; konstantes Delay nach Phasenschaltung
097A-   A2 11       LDX   #$11
097C-   CA          DEX
097D-   D0 FD       BNE   $097C
097F-   E9 01       SBC   #$01
0981-   D0 F7       BNE   $097A
0983-   A6 2B       LDX   $2B          ; Reload Slotno * 16
0985-   60          RTS                ; mit NE, d.h. Z-Flag gelöscht
```

**– READ BLOCK** (Nummer: $46/$47, Ziel: $44/$45)

```
0986-   A5 46       LDA   $46          ; Übersetzung von Blockno in Track/Sektor
0988-   29 07       AND   #$07
```

```
098A-  C9 04     CMP   #$04        ; 0   1   2   3
098C-  29 03     AND   #$03        ; - - - - - - - - - - - - - - - -
098E-  08        PHP                ; 0/2 4/6 8/A C/E 1/3 5/7 9/B D/F
098F-  0A        ASL
0990-  28        PLP
0991-  2A        ROL
0992-  85 3D     STA   $3D         ; Sektornummer gewünscht
0994-  A5 47     LDA   $47
0996-  4A        LSR
0997-  A5 46     LDA   $46         ; Blocknummer lo
0999-  6A        ROR                 wird zusammen mit Carry von Blockno hi
099A-  4A        LSR                 durch 8 dividiert. Diese Operation
099B-  4A        LSR                 funktioniert max. bis Blockno $1FF (511)
099C-  85 41     STA   $41         ; Tracknummer gewünscht
099E-  0A        ASL
099F-  85 51     STA   $51         ; Trackno gewünscht * 2 für ARM MOVER
09A1-  A5 45     LDA   $45
09A3-  85 27     STA   $27         ; Zieladresse hi
09A5-  A6 2B     LDX   $2B         ; Slotno * 16
09A7-  BD 89 C0  LDA   $C089,X     ; MOTOR ON - DRIVE SELECTed ist D1
09AA-  20 BC 09  JSR   $09BC       ; ARM MOVER und READ SECTOR
09AD-  E6 27     INC   $27         ; Zieladresse + $100
09AF-  E6 3D     INC   $3D         ; ein Sektor wird übersprungen, d.h
09B1-  E6 3D     INC   $3D           0=>2/1=>3 etc., s. Kommentar bei $098E
09B3-  B0 03     BCS   $09B8       ; ERROR
09B5-  20 BC 09  JSR   $09BC       ; ARM MOVER und READ des zweiten Sektors
09B8-  BC 88 C0  LDY   $C088,X     ; MOTOR OFF
09BB-  60        RTS               ; mit Carry set im Fehlerfall
```

## – ARM MOVER und READ SECTOR

```
09BC-  A5 40     LDA   $40         ; „Track jetzt" - vom letzten READ/vom Boot:
09BE-  0A        ASL
09BF-  85 53     STA   $53         ; OLDTRK * 2
09C1-  A9 00     LDA   #$00
09C3-  85 54     STA   $54

09C5-  A5 53     LDA   $53         ; „Track jetzt" * 2
09C7-  85 50     STA   $50
09C9-  38        SEC
09CA-  E5 51     SBC   $51         ; „Track gewünscht" * 2
09CC-  F0 14     BEQ   $09E2       ; sind gleich => READ SECTOR
09CE-  B0 04     BCS   $09D4
09D0-  E6 53     INC   $53         ; Track jetzt < Track gewünscht: INC
09D2-  90 02     BCC   $09D6
09D4-  C6 53     DEC   $53         ; Track jetzt > Track gewünscht: DEC
09D6-  38        SEC
09D7-  20 6D 09  JSR   $096D       ; PHASE ON in die gewünschte Richtung
09DA-  A5 50     LDA   $50         ; „alter Track"
09DC-  18        CLC
09DD-  20 6F 09  JSR   $096F       ; PHASE OFF für den „alten Track"
09E0-  D0 E3     BNE   $09C5       ; „always" - „LDX $2B/RTS" ist <> 0
```

## – ON CORRECT TRACK

```
09E2-  A0 7F     LDY   #$7F
09E4-  84 52     STY   $52         ; Init der Retries
09E6-  08        PHP
09E7-  28        PLP               ; Re-Entry für Retries
09E8-  38        SEC
09E9-  C6 52     DEC   $52
09EB-  F0 CE     BEQ   $09BB       ; RTS mit Carry gesetzt => ERROR
09ED-  18        CLC               ; Flag: Suche nach Address Field
09EE-  08        PHP
```

```
09EF-   88          DEY             ; Retry-Zähler für Warten auf nächstes
09F0-   F0 F5       BEQ   $09E7       Address Field
09F2-   BD 8C C0    LDA   $C08C,X
09F5-   10 FB       BPL   $09F2
--- ************************** ---   ; hier fängt die Kopie von der Card an
09F7-   49 D5       EOR   #$D5        ; $D5
09F9-   D0 F4       BNE   $09EF
09FB-   BD 8C C0    LDA   $C08C,X
09FE-   10 FB       BPL   $09FB
0A00-   C9 AA       CMP   #$AA        ; $AA
0A02-   D0 F3       BNE   $09F7
0A04-   EA          NOP
0A05-   BD 8C C0    LDA   $C08C,X
0A08-   10 FB       BPL   $0A05
0A0A-   C9 96       CMP   #$96        ; $96
0A0C-   F0 09       BEQ   $0A17       ; ok. Address Field gefunden: Lesen
0A0E-   28          PLP
0A0F-   90 D7       BCC   $09E8       ; Retry, wenn Address Field gewünscht (CLC)
0A11-   49 AD       EOR   #$AD
0A13-   F0 25       BEQ   $0A3A       ; Data Field gefunden: Lesen
0A15-   D0 D1       BNE   $09E8       ; irgendetwas anderes: Retry
```

### – READ Address Field

```
0A17-   A0 03       LDY   #$03        ; 3 „double Bytes" zu lesen
0A19-   85 40       STA   $40         ; hier bleibt zum Schluß die Trackno stehen
0A1B-   BD 8C C0    LDA   $C08C,X
0A1E-   10 FB       BPL   $0A1B
0A20-   2A          ROL
0A21-   85 3C       STA   $3C         ; Temp für even/odd decoding
0A23-   BD 8C C0    LDA   $C08C,X
0A26-   10 FB       BPL   $0A23
0A28-   25 3C       AND   $3C         ; Ergebnis: „Sektornummer gefunden" im Akku,
0A2A-   88          DEY                 „Tracknummer gefunden" in $0040, Volno wird
0A2B-   D0 EC       BNE   $0A19         übersprungen, Checksum/Trailer ignoriert
0A2D-   28          PLP               ; „Suche Address Field"-Indikator
0A2E-   C5 3D       CMP   $3D         ; „Sektornummer gewünscht"
0A30-   D0 B6       BNE   $09E8       ; nein, Retry und nächstes Address Field
0A32-   A5 40       LDA   $40         ; „Track gefunden"
0A34-   C5 41       CMP   $41         ; „Track gewünscht"
0A36-   D0 4B       BNE   $0A83       ; neuer ARM MOVER, Retry (JMP $09BC)
0A38-   B0 B4       BCS   $09EE       ; ok. Indikator jetzt „Data Field gewünscht"
```

### – READ Data Field

```
0A3A-   A0 56       LDY   #$56
0A3C-   84 3C       STY   $3C         ; Herunterzähler und Index
0A3E-   BC 8C C0    LDY   $C08C,X     ; 1.Teil: „Zusätzliche $56 Bytes" abwärts
0A41-   10 FB       BPL   $0A3E
0A43-   59 D6 02    EOR   $02D6,Y     ; Read Translation Table (s. $C607 ff.)
0A46-   A4 3C       LDY   $3C         ; Herunterzähler + Index
0A48-   88          DEY
0A49-   99 00 03    STA   $0300,Y     ; Zielbereich: $0355 bis $0300
0A4C-   D0 EE       BNE   $0A3C
0A4E-   84 3C       STY   $3C         ; = $00
0A50-   BC 8C C0    LDY   $C08C,X     ; 2. Teil ($100 Bytes) aufwärts
0A53-   10 FB       BPL   $0A50
0A55-   59 D6 02    EOR   $02D6,Y     ; Read Translation
0A58-   A4 3C       LDY   $3C
0A5A-   91 26       STA   ($26),Y     ; in den Zielbereich von READ
0A5C-   C8          INY
0A5D-   D0 EF       BNE   $0A4E
```

```
0A5F-   BC 8C C0    LDY   $C08C,X       ; Checksum
0A62-   10 FB       BPL   $0A5F
0A64-   59 D6 02    EOR   $02D6,Y
0A67-   D0 AC       BNE   $0A15         ; Retry, wenn die EOR-Kette nicht 00 ergibt
```

**– DENIBBLE**

```
0A69-   A0 00       LDY   #$00          ; in die $100 Bytes im Zielbereich werden
0A6B-   A2 56       LDX   #$56            jetzt wieder die in den „zusätzlichen"
0A6D-   CA          DEX                   $56 Bytes gespeicherten beiden LSBs
0A6E-   30 FB       BMI   $0A6B          hineingeschoben. Die „normalen" Bytes
0A70-   B1 26       LDA   ($26),Y        liegen in der Form „00xx xxxx" vor, die
0A72-   5E 00 03    LSR   $0300,X        „zusätzlichen" Bytes haben die Form
0A75-   2A          ROL                   „00 33 22 11", d.h. die LSBs für das
0A76-   5E 00 03    LSR   $0300,X        erste Drittel der „normalen" Bytes stehen
0A79-   2A          ROL                   als niederwertigste Bits im Byte.
0A7A-   91 26       STA   ($26),Y        Es ergibt sich xxxx xx11 für das erste,
0A7C-   C8          INY                   xxxx xx22 für das zweite und xxxx xx33
0A7D-   D0 EE       BNE   $0A6D          für das dritte Drittel der $100 Bytes.
  ********** Ende des vom Controller herunterkopierten Bereichs ********

0A7F-   A6 2B       LDX   $2B           ; diese sieben Bytes wurden angehängt:
0A81-   18          CLC                   Reload Slotno * 16 und RTS anstelle von
0A82-   60          RTS                   „JMP  $0801" sowie:
0A83-   4C BC 09    JMP   $09BC         ; Retry-„Verlängerung" => ARM MOVER

0A86..0BFF      ; nicht benutzt – beim Power-On undefiniert

0C00..13FF      ; Volume Directory (4 Blocks) – vom Urlader geladen
1400..1DFF      ; nicht benutzt – beim Power-On undefiniert
1E00..1FFF      ; Indexblock des Files ProDOS – vom Urlader geladen
2000..          ; ProDOS RELOCATOR und MLI
```

## 2.2.4. DER PRODOS-RELOCATOR

```
MLI:              $2D00 – $4DFF
Global Page 64k:  $4E00
Global Page 48k:  $4F00
Thunderclock:     $5000 – $507D
IRQ/RESET:        $5080 – $50FF
Der Bereich $5100-$51FF ist unbenutzt (Nullen)
DiskII Driver:    $5200 – $58FF
RAMDisk Driver:   $2C00 – $2C99
REBOOT:           $5900 – $61FF
```

Weitere (temporär) kopierte Routinen:

```
Von $2401 auf $0080-$00C7: SET MACHID
Von $2234 auf $0800-$096C: Lader für das „.SYSTEM"-Programm nach Installation des MLI

2000-   A5 43       LDA   $43           ; Slotno * 16 – einziger übernommener Wert!
2002-   8D 65 21    STA   $2165         ; Unitnummer für erstes ONLINE
2005-   20 48 24    JSR   $2448         ; CLS/ „PRODOS 1.0.1 - COPR"
2008-   AE 71 21    LDX   $2171         ; $2234 => $0800 - $096C
200B-   AC 72 21    LDY   $2172         ; $23A0 => $03F0 - $03FF
200E-   20 63 26    JSR   $2663         ; $000A - $000B: ($000A): „APPLE ÜÄ"-ptr
2011-   90 03       BCC   $2016         ; $2401 => $0080 - $00C7: Get MACHID
2013-   4C 2A 21    JMP   $212A         ; RELOCATION/CONFIGURATION ERR
2016-   A0 00       LDY   #$00
2018-   A9 FF       LDA   #$FF
201A-   8D FF BF    STA   $BFFF         ; enhält die Speicherstelle $BFFF RAM?
201D-   4D FF BF    EOR   $BFFF
```

```
2020-  38          SEC
2021-  D0 74       BNE   $2097
2023-  8D FF BF    STA   $BFFF
2026-  AD FF BF    LDA   $BFFF
2029-  D0 6C       BNE   $2097       ; „RELOCATION/CONFIGURATION ERROR"
202B-  AD 82 C0    LDA   $C082
202E-  20 B0 23    JSR   $23B0       ; Setzt MACHID ($0C), $0A/B „APPLE II"
2031-  A5 0C       LDA   $0C
2033-  29 20       AND   #$20        ; zurückgebrachtes MACHID:
                                       hat Maschine 64k RAM?
2035-  D0 10       BNE   $2047       ; Sprung, wenn ja
2037-  A2 03       LDX   #$03
2039-  BD 79 21    LDA   $2179,X     ; ansonsten werden die Startadressen der
203C-  9D 73 21    STA   $2173,X       MOVER-Parameterlisten auf „48k" gesetzt
203F-  CA          DEX
2040-  10 F7       BPL   $2039
2042-  A9 B8       LDA   #$B8        ; und $B8 anstelle von $F8 als Start für
2044-  8D 28 26    STA   $2628         die RWTB der DISKII
2047-  AE 73 21    LDX   $2173       ; $5080 => $FF80 - $FFFF: IRQ und  RESET
204A-  AC 74 21    LDY   $2174       ; $4E00 => $BF00 - $BFFF: Global Page
204D-  20 63 26    JSR   $2663       ; lösche $F100 - $F7FF: MLI data space
2050-  B0 45       BCS   $2097       ; $2D00 => $D000 - $F0FF: MLI
2052-  AD 82 C0    LDA   $C082       ; $5200 => $F800 - $FEFF: DISKII RWTB
```

für 48k:

$4F00 => $BF00 - $BFFF / Löschung von $B100-$B7FF / $2D00 => $9000 - $B100, Re-set aller absoluten Referenzen im Bereich $D000 bis $F7FF auf *-$4000 / Korrektur der Startadressen der $C0-$D3 COMMANDS um *-$4000 / Re-set aller absoluten Referenzen im Bereich $F800 bis $FFFF um *-$4000. Die Nibble-Tabellen (RDTRANS/WRTRANS) müssen dabei übersprungen werden.

```
2055-  AE C0 FB    LDX   $FBC0       ; II: 60 / II+: EA / IIe: EA / IIc: 00
2058-  E0 04       CPX   #$04
205A-  B0 1A       BCS   $2076       ; Sprung für II/II+/IIe
205C-  A5 0C       LDA   $0C
205E-  09 08       ORA   #$08        ; Setzen von Bit 3, wenn nicht II/II+/IIe
2060-  29 3F       AND   #$3F        ; und Reset von Bit 7
2062-  E0 00       CPX   #$00
2064-  D0 02       BNE   $2068
2066-  09 80       ORA   #$80        ; 00: setzen von Bit 7 und Bit 3
2068-  E0 01       CPX   #$01          01: setzen von Bit 6 und Bit 3
206A-  D0 02       BNE   $206E         02: lösche Bit 7, setze Bit 3
206C-  09 40       ORA   #$40          03: setze Bit 7, Bit 6 und Bit 3
206E-  E0 03       CPX   #$03
2070-  D0 02       BNE   $2074         für zukünftige Maschinen?
2072-  09 C0       ORA   #$C0
2074-  85 0C       STA   $0C         ; MACHID-für II/II+/IIe nicht verändert

2076-  AD 65 21    LDA   $2165       ; Setzen der Unitno im Parmblock für
2079-  8D 6C 21    STA   $216C         BLOCKREAD des „.SYSTEM"
207C-  8D 30 BF    STA   $BF30       ; „last recently used unit"
207F-  20 E6 24    JSR   $24E6       ; sucht alle Slots nach DEVICES ab
2082-  AD 6C 21    LDA   $216C                   und prüft „APPLE II"
2085-  8D 30 BF    STA   $BF30       ; nochmal Unitno - nie geändert
2088-  AD 8B C0    LDA   $C08B
208B-  AD 8B C0    LDA   $C08B       ; BANKSELECT Bank 1 + WRITE ENABLE
208E-  AE 75 21    LDX   $2175       ; $5000 => $F142 - $F1BE: CLOCK
2091-  AC 76 21    LDY   $2176         (Relocate auf Slotno der Clock (s.$24E6))
2094-  20 63 26    JSR   $2663       ; 48k: $5000 => $B142, Reloc auf *-$4000
2097-  B0 2D       BCS   $20C6         UND Slotnummer der Clock-Card
```

```
2099-  AD 98 BF    LDA   $BF98        ; MACHID - von $24E6 aus $000C kopiert
209C-  29 20       AND   #$20
209E-  C9 20       CMP   #$20
20A0-  D0 26       BNE   $20C8        ; Sprung, wenn nur 48k
20A2-  A9 CF       LDA   #$CF
20A4-  8D 04 BF    STA   $BF04
20A7-  A9 EE       LDA   #$EE         ; sonst Setzen des REBOOT-Vectors ($EECF)
20A9-  8D 05 BF    STA   $BF05          in der System Global Page
20AC-  AD 83 C0    LDA   $C083
20AF-  AD 83 C0    LDA   $C083        ; BANKSELECT: Bank 2 (+ WRITE ENABLE)
20B2-  AE 77 21    LDX   $2177
20B5-  AC 78 21    LDY   $2178        ; $5900 => $D100 - $D3FF
20B8-  20 63 26    JSR   $2663        ; REBOOT wird in die Bank 2 kopiert
20BB-  A9 EE       LDA   #$EE
20BD-  8D 00 D0    STA   $D000        ; BANKID für Bank 2
20C0-  AD 8B C0    LDA   $C08B
20C3-  AD 8B C0    LDA   $C08B        ; BANKSELECT Bank 1 (+ WRITE ENABLE)
20C6-  B0 62       BCS   $212A        ; ERR im MOVER - unmöglich (kein Relocate)
20C8-  AD 98 BF    LDA   $BF98        ; MACHID
20CB-  29 30       AND   #$30         ; 128k ?
20CD-  49 30       EOR   #$30
20CF-  D0 03       BNE   $20D4
20D1-  20 FF 28    JSR   $28FF          Wenn ja, Installation des RAMDisk driver,
                                        RAMDSK auf Slot 3, Drive 2 in DEVTBL
```

## - PRODOS RELOCATED

```
20D4-  20 00 BF    JSR   $BF00        ; ONLINE: holt den Namen des Units, von dem
20D7-  C5 64 21                         ProDOS geladen wurde
20DA-  D0 4E       BNE   $212A        ; ERROR
20DC-  AD 81 02    LDA   $0281        ; in den Puffer ab $0281
20DF-  29 0F       AND   #$0F         ; Namenslänge = 0 => ERR
20E1-  F0 47       BEQ   $212A
20E3-  18          CLC
20E4-  69 01       ADC   #$01
20E6-  8D 80 02    STA   $0280        ; Namenslänge + 1
20E9-  A9 2F       LDA   #$2F         ; und „/" als erstes Zeichen auf $0281
20EB-  8D 81 02    STA   $0281        ; => vollständiger Pathname
20EE-  20 00 BF    JSR   $BF00
20F1-  C6 68 21                       ; SET PREFIX mit ONLINE Puffer + „/"
20F4-  D0 34       BNE   $212A        ; ERR
20F6-  A2 00       LDX   #$00
20F8-  86 14       STX   $14
20FA-  A0 02       LDY   #$02         ; Block 00 02 (Voldir Key Block)
20FC-  A9 0C       LDA   #$0C         ; Pufferstart: $0C00
20FE-  85 15       STA   $15
2100-  8D 6E 21    STA   $216E        ; Zieladresse zum Parmblock von BLOCKREAD
2103-  8C 6F 21    STY   $216F        ; Blocknummer lo
2106-  8E 70 21    STX   $2170        ; Blocknummer hi
2109-  20 00 BF    JSR   $BF00
210C-  80 6B 21                       ; BLOCKREAD: Voldir
210F-  D0 19       BNE   $212A
2111-  A0 03       LDY   #$03
2113-  B1 14       LDA   ($14),Y
2115-  AA          TAX
2116-  88          DEY
2117-  11 14       ORA   ($14),Y      ; Forward Reference = 00 00?
2119-  F0 0C       BEQ   $2127        ; Sprung, wenn letzter Block der Voldir
211B-  B1 14       LDA   ($14),Y
211D-  A8          TAY                ; sonst Speichern der Forward Reference
211E-  A5 15       LDA   $15
2120-  18          CLC
2121-  69 02       ADC   #$02         ; Zieladresse + $200
```

```
2020-   38          SEC
2021-   D0 74       BNE   $2097
2023-   8D FF BF    STA   $BFFF
2026-   AD FF BF    LDA   $BFFF
2029-   D0 6C       BNE   $2097       ; „RELOCATION/CONFIGURATION ERROR"
202B-   AD 82 C0    LDA   $C082
202E-   20 B0 23    JSR   $23B0       ; Setzt MACHID ($0C), $0A/B „APPLE II"
2031-   A5 0C       LDA   $0C
2033-   29 20       AND   #$20        ; zurückgebrachtes MACHID:
                                        hat Maschine 64k RAM?
2035-   D0 10       BNE   $2047       ; Sprung, wenn ja
2037-   A2 03       LDX   #$03
2039-   BD 79 21    LDA   $2179,X     ; ansonsten werden die Startadressen der
203C-   9D 73 21    STA   $2173,X       MOVER-Parameterlisten auf „48k" gesetzt
203F-   CA          DEX
2040-   10 F7       BPL   $2039
2042-   A9 B8       LDA   #$B8        ; und $B8 anstelle von $F8 als Start für
2044-   8D 28 26    STA   $2628         die RWTB der DISKII
2047-   AE 73 21    LDX   $2173       ; $5080 => $FF80 - $FFFF: IRQ und  RESET
204A-   AC 74 21    LDY   $2174       ; $4E00 => $BF00 - $BFFF: Global Page
204D-   20 63 26    JSR   $2663       ; lösche $F100 - $F7FF: MLI data space
2050-   B0 45       BCS   $2097       ; $2D00 => $D000 - $F0FF: MLI
2052-   AD 82 C0    LDA   $C082       ; $5200 => $F800 - $FEFF: DISKII RWTB
```

für 48k:
$4F00 => $BF00 - $BFFF / Löschung von $B100-$B7FF / $2D00 => $9000 - $B100, Re-set aller absoluten Referenzen im Bereich $D000 bis $F7FF auf *-$4000 / Korrektur der Startadressen der $C0-$D3 COMMANDS um *-$4000 / Re-set aller absoluten Referenzen im Bereich $F800 bis $FFFF um *-$4000. Die Nibble-Tabellen (RDTRANS/WRTRANS) müssen dabei übersprungen werden.

```
2055-   AE C0 FB    LDX   $FBC0       ; II: 60 / II+: EA / IIe: EA / IIc: 00
2058-   E0 04       CPX   #$04
205A-   B0 1A       BCS   $2076       ; Sprung für II/II+/IIe
205C-   A5 0C       LDA   $0C
205E-   09 08       ORA   #$08        ; Setzen von Bit 3, wenn nicht II/II+/IIe
2060-   29 3F       AND   #$3F        ; und Reset von Bit 7
2062-   E0 00       CPX   #$00
2064-   D0 02       BNE   $2068
2066-   09 80       ORA   #$80        ; 00: setzen von Bit 7 und Bit 3
2068-   E0 01       CPX   #$01          01: setzen von Bit 6 und Bit 3
206A-   D0 02       BNE   $206E         02: lösche Bit 7, setze Bit 3
206C-   09 40       ORA   #$40          03: setze Bit 7, Bit 6 und Bit 3
206E-   E0 03       CPX   #$03
2070-   D0 02       BNE   $2074         für zukünftige Maschinen?
2072-   09 C0       ORA   #$C0
2074-   85 0C       STA   $0C         ; MACHID-für II/II+/IIe nicht verändert

2076-   AD 65 21    LDA   $2165       ; Setzen der Unitno im Parmblock für
2079-   8D 6C 21    STA   $216C         BLOCKREAD des „.SYSTEM"
207C-   8D 30 BF    STA   $BF30       ; „last recently used unit"
207F-   20 E6 24    JSR   $24E6       ; sucht alle Slots nach DEVICES ab
2082-   AD 6C 21    LDA   $216C                     und prüft „APPLE II"
2085-   8D 30 BF    STA   $BF30       ; nochmal Unitno - nie geändert
2088-   AD 8B C0    LDA   $C08B
208B-   AD 8B C0    LDA   $C08B       ; BANKSELECT Bank 1 + WRITE ENABLE
208E-   AE 75 21    LDX   $2175       ; $5000 => $F142 - $F1BE: CLOCK
2091-   AC 76 21    LDY   $2176         (Relocate auf Slotno der Clock (s.$24E6))
2094-   20 63 26    JSR   $2663       ; 48k: $5000 => $B142, Reloc auf *-$4000
2097-   B0 2D       BCS   $20C6         UND Slotnummer der Clock-Card
```

```
2099-  AD 98 BF   LDA   $BF98        ; MACHID - von $24E6 aus $000C kopiert
209C-  29 20      AND   #$20
209E-  C9 20      CMP   #$20
20A0-  D0 26      BNE   $20C8        ; Sprung, wenn nur 48k
20A2-  A9 CF      LDA   #$CF
20A4-  8D 04 BF   STA   $BF04
20A7-  A9 EE      LDA   #$EE         ; sonst Setzen des REBOOT-Vectors ($EECF)
20A9-  8D 05 BF   STA   $BF05          in der System Global Page
20AC-  AD 83 C0   LDA   $C083
20AF-  AD 83 C0   LDA   $C083        ; BANKSELECT: Bank 2 (+ WRITE ENABLE)
20B2-  AE 77 21   LDX   $2177
20B5-  AC 78 21   LDY   $2178        ; $5900 => $D100 - $D3FF
20B8-  20 63 26   JSR   $2663        ; REBOOT wird in die Bank 2 kopiert
20BB-  A9 EE      LDA   #$EE
20BD-  8D 00 D0   STA   $D000        ; BANKID für Bank 2
20C0-  AD 8B C0   LDA   $C08B
20C3-  AD 8B C0   LDA   $C08B        ; BANKSELECT Bank 1 (+ WRITE ENABLE)
20C6-  B0 62      BCS   $212A        ; ERR im MOVER - unmöglich (kein Relocate)
20C8-  AD 98 BF   LDA   $BF98        ; MACHID
20CB-  29 30      AND   #$30         ; 128k ?
20CD-  49 30      EOR   #$30
20CF-  D0 03      BNE   $20D4
20D1-  20 FF 28   JSR   $28FF          Wenn ja, Installation des RAMDisk driver,
                                       RAMDSK auf Slot 3, Drive 2 in DEVTBL
```

## – PRODOS RELOCATED

```
20D4-  20 00 BF   JSR   $BF00        ; ONLINE: holt den Namen des Units, von dem
20D7-  C5 64 21                        ProDOS geladen wurde
20DA-  D0 4E      BNE   $212A        ; ERROR
20DC-  AD 81 02   LDA   $0281        ; in den Puffer ab $0281
20DF-  29 0F      AND   #$0F         ; Namenslänge = 0 => ERR
20E1-  F0 47      BEQ   $212A
20E3-  18         CLC
20E4-  69 01      ADC   #$01
20E6-  8D 80 02   STA   $0280        ; Namenslänge + 1
20E9-  A9 2F      LDA   #$2F         ; und „/" als erstes Zeichen auf $0281
20EB-  8D 81 02   STA   $0281        ; => vollständiger Pathname
20EE-  20 00 BF   JSR   $BF00
20F1-  C6 68 21                      ; SET PREFIX mit ONLINE Puffer + „/"
20F4-  D0 34      BNE   $212A        ; ERR
20F6-  A2 00      LDX   #$00
20F8-  86 14      STX   $14
20FA-  A0 02      LDY   #$02         ; Block 00 02 (Voldir Key Block)
20FC-  A9 0C      LDA   #$0C         ; Pufferstart: $0C00
20FE-  85 15      STA   $15
2100-  8D 6E 21   STA   $216E        ; Zieladresse zum Parmblock von BLOCKREAD
2103-  8C 6F 21   STY   $216F        ; Blocknummer lo
2106-  8E 70 21   STX   $2170        ; Blocknummer hi
2109-  20 00 BF   JSR   $BF00
210C-  80 6B 21                      ; BLOCKREAD: Voldir
210F-  D0 19      BNE   $212A
2111-  A0 03      LDY   #$03
2113-  B1 14      LDA   ($14),Y
2115-  AA         TAX
2116-  88         DEY
2117-  11 14      ORA   ($14),Y      ; Forward Reference = 00 00?
2119-  F0 0C      BEQ   $2127        ; Sprung, wenn letzter Block der Voldir
211B-  B1 14      LDA   ($14),Y
211D-  A8         TAY                ; sonst Speichern der Forward Reference
211E-  A5 15      LDA   $15
2120-  18         CLC
2121-  69 02      ADC   #$02         ; Zieladresse + $200
```

```
2123-   C9 14       CMP   #$14          ; Loop bis VolDIR-Ende oder Ziel > $13FF
2125-   90 D7       BCC   $20FE            d.h. dem vierten Block

2127-   4C 00 08    JMP   $0800          ; VOLDIR geladen, nächste Stufe des Boot
```

**– ERR EXIT**

```
212A-   8D 82 C0    STA   $C082          ; ROMSELECT
212D-   20 58 FC    JSR   $FC58          ; HOME
2130-   A0 25       LDY   #$25
2132-   B9 3E 21    LDA   $213E,Y        ; „RELOCATION/CONFIGURATION ERROR"
2135-   99 A9 05    STA   $05A9,Y
2138-   88          DEY                    wird direkt in den Bildschirm kopiert
2139-   10 F7       BPL   $2132
213B-   4C 3B 21    JMP   $213B          ; „Aufhängen" bis RESET

213E- AA AA A0 D2 C5 CC CF C3  C1 D4 C9 CF CE A0 AF A0     ** RELOCATION /
214E- C3 CF CE C6 C9 C7 D5 D2  C1 D4 C9 CF CE A0 C5 D2     CONFIGURATION ER
215E- D2 CF D2 A0 AA AA  03 00 00                          ROR **
```

– Parameterblocks für MLI-Calls

```
2164- 02 60 81 02          ; erstes ONLINE: Unitno auf „Boot"-Drive gesetzt
2168- 01 02 80             ; SET PREFIX mit gefundenem Namen (ONLINE - Puffer + „/")
216B- 03 60 xx xx xx xx    ; BLOCKREAD: 00 02 auf $0C00 für READ der VolDIR, Zieladresse
                             und Blocknummer werden im Loop erhöht.
```

– Startadressen der MOVER-Parameterlisten für 64k:

```
2171- 7D 21     ; Parameterstart für ersten MOVER-Aufruf
2173- FF 21     ; zweiter Aufruf
2175- 21 22     ; dritter Aufruf
2177- 9A 21     ; vierter Aufruf (REBOOT)
```

– Startadressen für 48k – nach $2173 kopiert

```
2179- A2 21     ; Parameterstart für den zweiten Aufruf des MOVER
217A- E9 21     ; dritter Aufruf - erster Aufruf bleibt gleich, vierter entfällt
```

– Erster Aufruf des MOVER: 48 und 64k

```
217D- 01 00 08 6C 01 34 22          ; 01: MOVE $016C BYTES FROM $2234 TO $0800
2184- 01 F0 03 10 00 A0 23          ; 01: MOVE $0010 BYTES FROM $23A0 TO $03F0
218B- 01 0A 00 02 00 14 00          ; 01: MOVE $0002 BYTES FROM $0014 TO $000A
2192- 01 80 00 47 00 01 24          ; 01: MOVE $0047 BYTES FROM $2401 TO $0080
2199- FF                            ; STOP
```

– Vierter Aufruf des Mover nur für 64k: REBOOT

```
219A- 01 00 D1 00 03 00 59         ; 01: MOVE $0300 BYTES FROM $5900 TO $D100
21A1- FF                           ; STOP
```

– Zweiter Aufruf für 48k

```
21A2- 01 00 BF 00 01 00 4F          ; 01: MOVE $0100 BYTES FROM $4F00 TO $BF00
21A9- 00 B1 00 07 01                ; 00: CLEAR $B100 - $B206
21AE- 01 00 90 00 21 00 2D          ; 01: MOVE $2100 BYTES FROM $2D00 TO $9000
21B5- 04 00 90 CE 1E 00 90 00 D0 F7 C0 ; 04: CHANGE REFS TO $D0-$F0 IN $9000-$AECE
                                         TO *+$C0, i.e. *-$40
21C0- 03 65 AF 28 00 65 AF 00 D0 F0 C0 ; 03: CHANGE $C0-$D3 LOOKUP ohne Instr-Länge
```

```
21CB- 01 00 B8 00 07 00 52          ; 01: MOVE $0700 BYTES FROM $5200 TO $B800
21D4- 04 00 B8 95 01 00 B8 00 F8 FE C0 ; 04: CHANGE REFS TO $F8-$FE TO *-$40
21DF- 04 85 BB 39 03 85 BB 00 F8 FE C0 ; 04: CHANGE REFS TO $F8-$FE TO *-$40
21E8- FF                            ; STOP
```

– Dritter Aufruf für 48k: Thunderclock

```
21E9- 01 42 B1 7D 00 00 50          ; 01: MOVE $007D BYTES FROM $5000 TO $B142
21F0- 04 42 B1 69 00 42 B1 01 C1 F1; 04: CHANGE REFS TO $C1-$F1 IN RANGE + SLOT
      C1 F1 ** C0                           (** = Slotno)
21FE- FF                            ; STOP
```

– Zweiter Aufruf für 64k

```
21FF- 01 80 FF 80 00 80 50          ; 01: MOVE $0080 BYTES FROM $5080 TO $FF80
2206- 01 00 BF 00 01 00 4E          ; 01: MOVE $0100 BYTES FROM $4E00 TO $BF00
220D- 00 00 F1 00 07                   ; 00: CLEAR $F100 - $F7FF
2212- 01 00 D0 00 21 00 2D          ; 01: MOVE $2100 BYTES FROM $2D00 TO $D000
2219- 01 00 F8 00 07 00 52          ; 01: MOVE $0700 BYTES FROM $5200 TO $F800
2220- FF                               ; STOP
```

- Dritter Aufruf für 64k: Thunderclock

```
2221- 01 42 F1 7D 00 00 50          ; 01: MOVE $007D BYTES FROM $5000 TO $F142
2228- 04 42 F1 69 00 42 F1 00 C1 C1 ** ; 04: ADAPT $F142-+69 TO SLOTNO FOUND
2233- FF                            ; STOP
```

– Letzte Stufe des Boot: Lader für das „.SYSTEM"-Programm. Auf $0800 kopiert!

```
2234-   A9 0C       LDA   #$0C          ; Sprung hierher mit geladener VolDIR
2236-   85 11       STA   $11
2238-   A9 04       LDA   #$04          ; Start mit erstem Eintrag nach $0C04
223A-   ** 2C **                          von $223A: BIT $10A5
223B-   A5 10       LDA   $10           ; d.h. „LDA $10" ist maskiert
223D-   18          CLC
223E-   6D 23 0C    ADC   $0C23         ; + Länge eines Eintrags => NEXT Eintrag
2241-   85 10       STA   $10
2243-   B0 12       BCS   $2257         ; $0010/11 auf nächsten Eintrag
2245-   6D 23 0C    ADC   $0C23         ; Test: überschreitet dieser nächste Eintrag
2248-   90 0F       BCC   $2259           die Directory-Länge, d.h. ist die
224A-   A5 11       LDA   $11             Volume-Directory hier zuende?
224C-   4A          LSR
224D-   90 0A       BCC   $2259         ; Speicherseite ist gerade: Blockanfang
224F-   C9 09       CMP   #$09          ; Speicherseite ungerade: $1400 erreicht?
2251-   F0 1E       BEQ   $2271         ; „UNABLE TO FIND A .SYSTEM FILE"
2253-   A9 04       LDA   #$04          ; nächster Block: Start auf $xx04
2255-   85 10       STA   $10
2257-   E6 11       INC   $11
2259-   A0 10       LDY   #$10          ; Byte $10 im File-Entry: File Type
225B-   A9 FF       LDA   #$FF
225D-   51 10       EOR   ($10),Y
225F-   D0 DA       BNE   $223B         ; kein SYS file: NEXT

2261-   A8          TAY                 ; = 0
2262-   B1 10       LDA   ($10),Y       ; Byte $00 of: Storage Type + Namenslänge
2264-   F0 D5       BEQ   $223B         ; NEXT, wenn File deleted
2266-   29 0F       AND   #$0F
2268-   8D 80 02    STA   $0280         ; Namenslänge
226B-   C9 08       CMP   #$08          ; „.SYSTEM" = 7 Zeichen
226D-   90 CC       BCC   $223B         ; kürzer als 8 Zeichen: NEXT
226F-   B0 03       BCS   $2274         ; sonst Vergleich
```

```
2271-   F0 6E       BEQ   $22E1          ; „Verlängerung" für Fehler ($2251)
2273-   00          BRK

2274-   A8          TAY                  ; Namensvergleich von hinten nach vorne:
2275-   A2 06       LDX   #$06
2277-   B1 10       LDA   ($10),Y
2279-   5D 65 09    EOR   $0965,X        ; „.SYSTEM" - (hier auf $2399,X)
227C-   0A          ASL                  ; MSB wird maskiert
227D-   D0 BC       BNE   $223B          ; endet nicht auf „.SYSTEM": NEXT
227F-   88          DEY
2280-   CA          DEX
2281-   10 F4       BPL   $2277
2283-   A0 00       LDY   #$00           ; ok. Name endet mit „.SYSTEM"
2285-   C8          INY
2286-   B1 10       LDA   ($10),Y
2288-   99 80 02    STA   $0280,Y        ; Kopie des Namens in den Parmblock mit
228B-   09 80       ORA   #$80           ; gesetztem MSB
228D-   99 3B 09    STA   $093B,Y        ; (hier auf $236F,X)
2290-   CC 80 02    CPY   $0280
2293-   D0 F0       BNE   $2285          ; Loop bis Namensende
2295-   A9 A0       LDA   #$A0
2297-   99 3C 09    STA   $093C,Y        ; ein Space wird angehängt (hier: $2370,X)
229A-   98          TYA
229B-   69 13       ADC   #$13
229D-   8D 4F 09    STA   $094F          ; (hier auf $2383)
22A0-   20 00 BF    JSR   $BF00
22A3-   C8 50 09                         ; OPEN: (Parmblock hier auf $2384)
22A6-   D0 46       BNE   $22EE          ; Name: $0280 / Puffer: $1400
22A8-   20 00 BF    JSR   $BF00
22AB-   D1 56 09                         ; GET EOF: (Parmblock hier auf $238A)
22AE-   D0 3E       BNE   $22EE          ; „UNABLE TO LOAD ... SYSTEM"
22B0-   AD 5A 09    LDA   $095A          ; MSByte der File-Länge (hier auf $238E)
22B3-   D0 53       BNE   $2308          ; „SYSTEM PROGRAM TOO LARGE"
22B5-   AD 59 09    LDA   $0959          ; mittleres Byte (hier auf $238D)
22B8-   C9 98       CMP   #$98
22BA-   B0 4C       BCS   $2308          ; „SYSTEM PROGRAM TOO LARGE"
22BC-   8D 60 09    STA   $0960          ; der EOF wird als Request Count für READ
22BF-   AD 58 09    LDA   $0958            gesetzt (2 Bytes)
22C2-   8D 5F 09    STA   $095F
22C5-   20 00 BF    JSR   $BF00
22C8-   CA 5B 09                         ; READ (Parmblock hier auf $238F)
22CB-   F0 06       BEQ   $22D3            Startadresse: $2000
22CD-   C9 56       CMP   #$56           ; BAD BUFFER ADDRESS?
22CF-   F0 37       BEQ   $2308          ; wenn ja, „SYSTEM PROGRAM TOO LARGE"
22D1-   D0 1B       BNE   $22EE          ; sonst „UNABLE TO LOAD ....SYSTEM"
22D3-   20 00 BF    JSR   $BF00
22D6-   CC 63 09                         ; keine Fehler: CLOSE (Parmblock hier $2397)
22D9-   D0 13       BNE   $22EE
22DB-   AD 82 C0    LDA   $C082          ; ROMON
22DE-   4C 00 20    JMP   $2000          ; Sprung zum „.SYSTEM"-Programm
```

## – Error Exits

```
22E1-   A2 27       LDX   #$27
22E3-   BD E2 08    LDA   $08E2,X        ; hier: $2316,X
22E6-   9D A8 07    STA   $07A8,X        ; ** UNABLE TO FIND A „.SYSTEM" FILE **
22E9-   CA          DEX
22EA-   10 F7       BPL   $22E3
22EC-   30 25       BMI   $2313          ; „Aufhängen" bis RESET

22EE-   AC 4F 09    LDY   $094F          ; hier: $2383
22F1-   A9 27       LDA   #$27
22F3-   38          SEC
```

```
22F4-   ED 4F 09    SBC   $094F         ; Holen des Filenamens und Zentrierung
22F7-   4A          LSR                   auf dem Bildschirm (nett...)
22F8-   6D 4F 09    ADC   $094F
22FB-   AA          TAX
22FC-   B9 2A 09    LDA   $092A,Y       ; hier: $235E,Y
22FF-   9D A8 07    STA   $07A8,X       ; ** UNABLE TO LOAD xxxx.SYSTEM **
2302-   CA          DEX
2303-   88          DEY
2304-   10 F6       BPL   $22FC
2306-   30 0B       BMI   $2313         ; „Aufhängen" bis RESET

2308-   A0 1E       LDY   #$1E
230A-   B9 0A 09    LDA   $090A,Y       ; hier: $233E,Y
230D-   99 AD 07    STA   $07AD,Y       ; ** SYSTEM PROGRAM TOO LARGE **
2310-   CA          DEX
2311-   10 F7       BPL   $230A

2313-   30 FE       BMI   $2313         ; endgültiger „Aufhänger"


2316- AA AA A0 A0 D5 CE C1 C2  CC C5 A0 D4 CF A0 C6 C9    **  UNABLE TO FI
2326- CE C4 A0 C1 A0 A2 AE D3  D9 D3 D4 C5 CD A2 A0 C6    ND A „.SYSTEM" F
2336- C9 CC C5 A0 A0 AA AA A0                            ILE **
233E- AA AA A0 A0 D3 D9 D3 D4  C5 CD A0 D0 D2 CF C7 D2    ** SYSTEM PROGRA
234F- CD A0 D4 CF CF A0 CC C1  D2 C7 C5 A0 A0 AA AA      M TOO LARGE  **
235E- AA AA A0 D5 CE C1 C2 CC  C5 A0 D4 CF A0 CC CF C1    ** UNABLE TO LOA
236E- C4 A0 D8 AE D3 D9 D3 D4  C5 CD A0 AA AA AA AA AA    D X.SYSTEM *****
237E- AA AA AA AA AA                                     *****

2383- 00                          ; Scratch zur Längenberechnung des Filenamens

2384- 03  8002 0014 01            ; OPEN: Pathname $0280, Puffer $1400
238A- 02 01  000000               ; GET EOF: Refno immer 01, 3 Byte zurück
238F- 04 01 0020 0000 0000        ; READ: Refno 01, Req.Count von EOF kopiert
2397- 01 00                       ; CLOSE: ALL

2399- AE D3 D9 D3  D4 C5 CD                              .SYSTEM
```

******* ENDE DES KOPIERTEN BEREICHS*******

```
23A0- 59 FA 59 FF 5A 4C 59 FF 4C 59 FF 4C 59 FF EB BF     ; Page 3 - Vektoren
```

## - WHAT MACHINE?

```
23B0-   A9 00       LDA   #$00
23B2-   85 0C       STA   $0C           ; MACHID auf $00
23B4-   AE B3 FB    LDX   $FBB3
23B7-   E0 38       CPX   #$38          ; II: MACHID $00
23B9-   F0 1F       BEQ   $23DA
23BB-   A9 80       LDA   #$80
23BD-   E0 06       CPX   #$06          ; IIe: MACHID $80
23BF-   F0 19       BEQ   $23DA
23C1-   A9 40       LDA   #$40
23C3-   E0 EA       CPX   #$EA          ; II+: MACHID $40
23C5-   D0 0D       BNE   $23D4         ; ERROR
23C7-   AE 1E FB    LDX   $FB1E         ; II+: PREAD (PADDLE READ) wird
23CA-   E0 AD       CPX   #$AD            auch geprüft - Unterscheidung zu III
23CC-   F0 0C       BEQ   $23DA         ; ok. ist II+
23CE-   A9 D0       LDA   #$D0          ; sonst APPLE III in emulation mode
23D0-   E0 8A       CPX   #$8A
23D2-   F0 2A       BEQ   $23FE         ; Sprung zum Test auf „128k"
```

```
23D4-   A9 02       LDA   #$02          ; demoliert die erste Instr.: $(14) = $0080
23D6-   91 14       STA   ($14),Y       ; Y = 0
23D8-   D0 24       BNE   $23FE         ; JMP nach $0080

23DA-   85 0C       STA   $0C           ; MACHID momentan
23DC-   AE 8B C0    LDX   $C08B
23DF-   AE 8B C0    LDX   $C08B
23E2-   A9 AA       LDA   #$AA
23E4-   8D 00 D0    STA   $D000         ; Test der Language Card
23E7-   4D 00 D0    EOR   $D000
23EA-   D0 0E       BNE   $23FA         ; nur 48k
23EC-   4E 00 D0    LSR   $D000
23EF-   A9 55       LDA   #$55
23F1-   4D 00 D0    EOR   $D000
23F4-   D0 04       BNE   $23FA         ; nur 48k
23F6-   A9 20       LDA   #$20          ; ok.: 64k
23F8-   D0 02       BNE   $23FC
23FA-   A9 10       LDA   #$10          ; keine RAM Card: 48k
23FC-   05 0C       ORA   $0C

23FE-   4C 80 00    JMP   $0080         ; Test „128k", setzt MACHID und $0A/$0B
```

– diese Routine wird auf $0080 kopiert

```
2401-   85 0C       STA   $0C           ; MACHID
2403-   10 35       BPL   $243A         ; APPLE III: kein Test auf 128k
2405-   A9 EE       LDA   #$EE
2407-   8D 05 C0    STA   $C005
240A-   8D 03 C0    STA   $C003
240D-   8D 00 0C    STA   $0C00
2410-   8D 00 08    STA   $0800
2413-   AD 00 0C    LDA   $0C00
2416-   C9 EE       CMP   #$EE
2418-   D0 0E       BNE   $2428         ; keine extended 80-Col Card
241A-   0E 00 0C    ASL   $0C00
241D-   0A          ASL
241E-   CD 00 0C    CMP   $0C00
2421-   D0 05       BNE   $2428         ; nein
2423-   CD 00 08    CMP   $0800
2426-   D0 03       BNE   $242B         ; ok. hat 128k
2428-   38          SEC
2429-   B0 01       BCS   $242C
242B-   18          CLC
242C-   8D 04 C0    STA   $C004
242F-   8D 02 C0    STA   $C002
2432-   B0 06       BCS   $243A         ; C gesetzt, wenn keine RAMDisk
2434-   A5 0C       LDA   $0C
2436-   09 30       ORA   #$30          ; 128k (64k RAMDisk)
2438-   85 0C       STA   $0C
243A-   A5 0B       LDA   $0B
243C-   38          SEC
243D-   E9 05       SBC   #$05          ; Setzen von $0A/$0B auf $FB09
243F-   85 0B       STA   $0B
2441-   A5 0A       LDA   $0A
2443-   E9 00       SBC   #$00
2445-   85 0A       STA   $0A           ; $FB09: „APPLE II"
2447-   60          RTS
```

**– TITLE PAGE**

```
2448-   AD 30 C0    LDA   $C030
244B-   8D 0C C0    STA   $C00C         ; Abschalten der extended 80-Col Card
244E-   8D 00 C0    STA   $C000
```

```
2451-   20 84 FE    JSR   $FE84          ; SETNORM (nicht INVERSE)
2454-   20 2F FB    JSR   $FB2F          ; INIT (Textfenster auf volle Schirmgröße)
2457-   20 93 FE    JSR   $FE93          ; SETVID (PR#0)
245A-   20 89 FE    JSR   $FE89          ; SETKBD (IN#0)
245D-   D8          CLD
245E-   78          SEI
245F-   20 58 FC    JSR   $FC58          ; HOME
2462-   A2 07       LDX   #$07
2464-   BD 92 24    LDA   $2492,X
2467-   9D B6 04    STA   $04B6,X        ; „APPLE II"
246A-   CA          DEX
246B-   10 F7       BPL   $2464
246D-   A2 16       LDX   #$16
246F-   BD 9A 24    LDA   $249A,X
2472-   9D AE 05    STA   $05AE,X        ; „PRODOS 1.0.1 1-JAN-84"
2475-   CA          DEX
2476-   10 F7       BPL   $246F
2478-   A2 0D       LDX   #$0D
247A-   BD B1 24    LDA   $24B1,X
247D-   9D B3 06    STA   $06B3,X        ; „    (Leerzeichen)    "
2480-   CA          DEX
2481-   10 F7       BPL   $247A
2483-   A2 26       LDX   #$26
2485-   BD BF 24    LDA   $24BF,X
2488-   9D D0 07    STA   $07D0,X        ; „COPYRIGHT APPLE COMPUTER.."
248B-   CA          DEX
248C-   10 F7       BPL   $2485
248E-   AD 30 C0    LDA   $C030
2491-   60          RTS

2492- C1 D0 D0 CC C5 A0 DD DB  D0 D2 CF C4 CF D3 A0 B1    APPLE ÜÄPRODOS 1
24A2- AE B0 AE B1 A0 A0 A0 B1  AD CA C1 CE AD B8 B4 A0    .0.1   1-JAN-84
24B2- A0 A0 A0 A0 A0 A0 A0 A0  A0 A0 A0 A0 A0 C3 CF D0                 COP
24C2- D9 D2 C9 C7 C8 D4 A0 C1  D0 D0 CC C5 A0 C3 CF CD    YRIGHT APPLE COM
24D2- D0 D5 D4 C5 D2 AC A0 C9  CE C3 AE AC A0 B1 B9 B8    PUTER, INC., 198
24E2- B3 AD B8 B4                                        3-84
```

– Absuchen der Slots nach „bekannten" Karten (80-Z/CLOCK/CONTROLLER/ROMS)

```
24E6-   A2 00       LDX   #$00
24E8-   86 14       STX   $14
24EA-   86 15       STX   $15
24EC-   86 10       STX   $10
24EE-   CA          DEX
24EF-   8E 31 BF    STX   $BF31          ; DEVNUM auf $FF (keine DEVICES)
24F2-   A9 C7       LDA   #$C7
24F4-   85 11       STA   $11            ; $10/$11: Slotadrese, Start mit $C700
```

– NEXT SLOT

```
24F6-   AD FF CF    LDA   $CFFF          ; CLRROM – Abschalten aller Zusatz-ROMs
24F9-   A0 05       LDY   #$05
24FB-   B1 10       LDA   ($10),Y
24FD-   D9 29 26    CMP   $2629,Y        ; DISKID: $20 xx $00 xx $03
2500-   38          SEC
2501-   D0 66       BNE   $2569          ; Carry gesetzt: kein Disk-Controller hier
2503-   88          DEY
2504-   88          DEY
2505-   10 F4       BPL   $24FB          ; ok. Disk-Controller gefunden
2507-   B1 10       LDA   ($10),Y        ; Y = $FF => $CxFF
2509-   F0 20       BEQ   $252B          ; Sprung für DISKII
250B-   C9 FF       CMP   #$FF
250D-   B0 5A       BCS   $2569          ; Start der On-Card RWTB (lo) > $CxFE?
```

```
250F-  48          PHA             ; RWTB START - Lobyte
2510-  88          DEY
2511-  B1 10       LDA   ($10),Y   ; DEVICE CHARACTERISTICS
2513-  48          PHA
2514-  29 03       AND   #$03      ; muß mindestens „Status"
2516-  C9 03       CMP   #$03        und „Readable" sein (Bit 1/0 gesetzt)
2518-  F0 05       BEQ   $251F
251A-  68          PLA
251B-  68          PLA
251C-  38          SEC             ; irgendetwas anderes: SEC => keine Disk
251D-  D0 4A       BNE   $2569     ; „always"
251F-  68          PLA             ; Characteristics
2520-  4A          LSR
2521-  4A          LSR
2522-  4A          LSR             ; ins untere Nibble
2523-  4A          LSR
2524-  85 12       STA   $12       ; und Speichern
2526-  4A          LSR             ; setzt Carry, wenn 2 Drives möglich
2527-  A5 11       LDA   $11
2529-  D0 0A       BNE   $2535     ; „always"

252B-  85 12       STA   $12       ; „Special" für DISKII: Characteristics = 0
252D-  AD 27 26    LDA   $2627     ; Start der RWTB (lo) = 00
2530-  48          PHA
2531-  AD 28 26    LDA   $2628     ; Start der RWTB (hi) $F8 oder $B8 (48k)
2534-  38          SEC             ; Flag: 2 Drives per Controller möglich

2535-  48          PHA             ; RWTB-Startadresse hi ($Cx oder $F8/$B8)
2536-  08          PHP             ; „Push Carry" - 1/2 Drives möglich
2537-  A5 11       LDA   $11       ; Slotadresse hi ($Cx)
2539-  29 07       AND   #$07
253B-  0A          ASL
253C-  A8          TAY             ;  => Index in DEVTBL
253D-  0A          ASL
253E-  0A          ASL
253F-  0A          ASL             ; Shift ins höhere Nibble
2540-  05 12       ORA   $12       ; + Characteristics
2542-  AE 31 BF    LDX   $BF31     ; DEVNUM - Anzahl der DEVICEs - 1 (Start FF)
2545-  E8          INX
2546-  9D 32 BF    STA   $BF32,X   ; Eintrag in der DEVICE TABLE
2549-  0A          ASL             ; Osss 0000 => sss 0000 0
254A-  28          PLP
254B-  6A          ROR             ; Carry von „1/2 Drives" => Dsss 0000
254C-  10 04       BPL   $2552     ; wenn gesetzt: 2 Drives möglich
254E-  E8          INX
254F-  9D 32 BF    STA   $BF32,X   ; Eintrag für Drive 2 DEVICE TABLE
2552-  8E 31 BF    STX   $BF31     ; und Setzen der neuen DEVNUM
2555-  0A          ASL             ; bei 2 Drives ist das Carry wieder gesetzt
2556-  68          PLA
2557-  99 11 BF    STA   $BF11,Y   ; DRIVER START HIBYTE DRIVE 1
255A-  90 08       BCC   $2564     ; Sprung, wenn nur 1 Drive möglich
255C-  99 21 BF    STA   $BF21,Y   ; DRIVER START HIBYTE DRIVE 2
255F-  68          PLA
2560-  48          PHA
2561-  99 20 BF    STA   $BF20,Y   ; LOBYTE DRIVER START DRIVE 2
2564-  68          PLA
2565-  99 10 BF    STA   $BF10,Y   ; LOBYTE DRIVER START DRIVE 1
2568-  18          CLC             ; Flag: Disk gefunden, keine weiteren Tests
```

– Sprung hierher mit SEC, wenn kein Controller gefunden

```
2569-  20 B6 25    JSR   $25B6     ; wenn keine Disk: nur Eintrag in ROMBIT,
                                     sonst Test auf andere Karten
```

```
256C-   C6 11       DEC   $11          ; $Cx
256E-   A5 11       LDA   $11
2570-   29 07       AND   #$07
2572-   D0 82       BNE   $24F6        ; NEXT Slot, bis (inklusive) Slot 1
```

– Alle Slots abgesucht

Die CHARACTERISTICS TABLE wird so sortiert, daß der Drive, von dem ProDOS geladen wurde, als höchstes DEVICE in der Table steht. Bei einem ONLINE (intern oder explizit) wird dieser Drive als erster getestet (Ausnahme: RAMDisk, sie steht immer als oberstes, d.h. erstes DEVICE in der Tabelle, weil hier die Zugriffszeit am kürzesten ist).

```
2574-   A8          TAY                ; = 0
2575-   AE 31 BF    LDX   $BF31        ; DEVNUM
2578-   BD 32 BF    LDA   $BF32,X      ; Loop abwärts durch alle DEVICEs und „Push"
257B-   48          PHA                  aller DEVICES, die nicht im „Boot"-Slot
257C-   29 7F       AND   #$7F           sind.
257E-   4D 30 BF    EOR   $BF30
2581-   0A          ASL                ; die Drive-Nummer wird nicht berücksichtigt
2582-   D0 02       BNE   $2586
2584-   68          PLA                ; „Boot"-slot, wieder herunter vom Stack
2585-   C8          INY                ; Zähler für „Boot"-Slot DEVICES
2586-   CA          DEX
2587-   10 EF       BPL   $2578        ; Loop
2589-   AE 31 BF    LDX   $BF31        ; DEVNUM
258C-   98          TYA                ; Zähler für wieder „heruntergeworfene" D.
258D-   F0 14       BEQ   $25A3        ; Beim Boot von nirgendwo => Restore (aua!)
258F-   AD 30 BF    LDA   $BF30        ; = Boot slot
2592-   9D 32 BF    STA   $BF32,X      ; Eintrag auf der obersten Position
2595-   CA          DEX
2596-   30 12       BMI   $25AA        ; Sprung, wenn dies das einzige DEVICE war
2598-   88          DEY
2599-   30 08       BMI   $25A3        ; Sprung, wenn nur ein DEVICE im „Boot"-Slot
259B-   49 80       EOR   #$80         ; sonst wird hier fröhlich ein zweiter
259D-   9D 32 BF    STA   $BF32,X        Drive eingetragen! (Nur „sauber" für Boot
25A0-   CA          DEX                  von einem „zwei-Drive-Controller")
25A1-   30 07       BMI   $25AA
25A3-   68          PLA                ; Restore: wenn andere DEVICES gespeichert
25A4-   9D 32 BF    STA   $BF32,X      ; wurden, werden sie wieder in absteigender
25A7-   CA          DEX                ; Reihenfolge in die DEVTBL eingetragen

25A8-   10 F9       BPL   $25A3

25AA-   20 39 26    JSR   $2639        ; Prüft „APPLE II"
25AD-   F0 04       BEQ   $25B3        ; Arrgh!
25AF-   8D 98 BF    STA   $BF98        ; RTS mit MACHID im Akku => MACHID auf
25B2-   60          RTS                  der System Global Page

25B3-   4C D4 23    JMP   $23D4        ; demoliert $0080 und springt dann dahin
```

– Absuchen der Slots nach anderen bekannten Karten

```
25B6-   90 60       BCC   $2618        ; Disk gefunden: nur Eintrag in  ROMBIT
```

– Test auf Thunderclock

```
25B8-   A0 06       LDY   #$06
25BA-   B1 10       LDA   ($10),Y      ; $10/$11: $Cx00
25BC-   D9 29 26    CMP   $2629,Y      ; $08 $28 $58 $70 (PHP/PLP/CLI/BVS...)
25BF-   D0 1D       BNE   $25DE        ; keine Thunderclock bzw. Kompatible
25C1-   88          DEY
25C2-   88          DEY
```

```
25C3-  10 F5       BPL   $25BA        ; ok. Thunderclock gefunden
25C5-  A5 11       LDA   $11
25C7-  E9 C1       SBC   #$C1
25C9-  8D FC 21    STA   $21FC        ; RELOCATION FACTOR FÜR CLOCK ROUTINE
25CC-  8D 32 22    STA   $2232        ; $21FC: 48k; $2232: 64k – Liste für MOVER
25CF-  A9 4C       LDA   #$4C         ; Installation des JMP in der SGP für

25D1-  8D 06 BF    STA   $BF06          $82 – SET TIME (default: RTS)
25D4-  A5 0C       LDA   $0C          ; MACHID jetzt – MUSS inzwischen <> 0 sein!
25D6-  F0 D2       BEQ   $25AA        ; s. „Spezielles"
25D8-  09 01       ORA   #$01         ; set: HAS CLOCK
25DA-  85 0C       STA   $0C
25DC-  D0 3A       BNE   $2618        ; Eintrag dieses Slots in ROMBIT
```

– Test für 80 Col Card – nur APPLE, nicht VIDEX o.ä.!

```
25DE-  A0 05       LDY   #$05
25E0-  B1 10       LDA   ($10),Y
25E2-  C9 38       CMP   #$38         ; $Cx05: $38
25E4-  D0 21       BNE   $2607        ; nein
25E6-  A0 07       LDY   #$07
25E8-  B1 10       LDA   ($10),Y
25EA-  C9 18       CMP   #$18         ; $Cx07: $18
25EC-  D0 19       BNE   $2607        ; no
25EE-  A0 0B       LDY   #$0B
25F0-  B1 10       LDA   ($10),Y
25F2-  C9 01       CMP   #$01         ; $Cx0B: $01
25F4-  D0 11       BNE   $2607        ; non
25F6-  C8          INY
25F7-  B1 10       LDA   ($10),Y
25F9-  C9 88       CMP   #$88         ; $Cx0C: $88
25FB-  D0 0A       BNE   $2607        ; njet
25FD-  A5 0C       LDA   $0C          ; MACHID jetzt, s. $25D4
25FF-  F0 A9       BEQ   $25AA
2601-  09 02       ORA   #$02
2603-  85 0C       STA   $0C          ; setzt: HAS 80 COL CARD
2605-  D0 11       BNE   $2618        ; Eintrag dieses Slots in ROMBIT
```

– kein „bekanntes" ROM (d.h. Disk/Clock/80 Col) gefunden

```
2607-  A0 00       LDY   #$00
2609-  A2 00       LDX   #$00         ; Test dieses Slots auf konstanten READ,
260B-  B1 10       LDA   ($10),Y
260D-  C9 FF       CMP   #$FF         ; $FF auf Byte $00 => leerer Slot
260F-  F0 15       BEQ   $2626
2611-  D1 10       CMP   ($10),Y      ; 256 Reads von $Cx0Q
2613-  D0 11       BNE   $2626        ; ändert sich der Wert => leer
2615-  E8          INX                  Mit diesem Test werden auch RAM-Karten
2616-  D0 F9       BNE   $2611          in ROMBIT eingetragen
```

– Eintrag der Slotnummer in ROMBIT

```
2618-  A5 11       LDA   $11          ; $Cx
261A-  29 07       AND   #$07
261C-  AA          TAX
261D-  BD 31 26    LDA   $2631,X      ; $02/$04/$08/$10/$20/$40/$80 Bit Mask
2620-  0D 99 BF    ORA   $BF99
2623-  8D 99 BF    STA   $BF99        ; Bit gesetzt: ROM/RAM im Slot
2626-  60          RTS


2627- 00 F8                   ; RWTB-Startadresse für DISKII. $B800 für 48k
2629- ** 20 ** 00 ** 03       ; DISKID
```

```
      08 ** 28 ** 58 ** 70      ; ID für THUNDERCLOCK
2630- 3C                        ; viertes Byte von DISKID - nicht geprüft
2631- 00                        ; Scratch für den „APPLEII"-Test
2632- 02 04 08 10 20 40 80      ; Bit Mask für ROMBIT - kein Bit für Slot 0
```

**– VERIFY „APPLEII"** auf $FB09...

```
2639-  18          CLC
263A-  AC 31 26    LDY   $2631          ; = 0
263D-  B1 0A       LDA   ($0A),Y        ; $000A/B sind auf $FB09 gesetzt
263F-  29 DF       AND   #$DF           ; Übersetzung auf Großschreibung
2641-  6D 31 26    ADC   $2631
2644-  8D 31 26    STA   $2631
2647-  2E 31 26    ROL   $2631
264A-  C8          INY
264B-  CC 34 26    CPY   $2634          ; = $08
264E-  D0 ED       BNE   $263D
2650-  98          TYA                  ; = $08
2651-  0A          ASL
2652-  0A          ASL
2653-  0A          ASL
2654-  0A          ASL
2655-  A8          TAY                  ; = $80
2656-  4D 31 26    EOR   $2631          ; muß $F5 ergeben
2659-  69 0B       ADC   #$0B           ; + $0B = 0, Carry gesetzt
265B-  D0 03       BNE   $2660
265D-  A5 0C       LDA   $0C            ; nach RTS: STA MACHID
265F-  60          RTS
2660-  A9 00       LDA   #$00           ; „Error" Exit: Setzt MACHID = 0
2662-  60          RTS
```

**– MOVER** – kopiert und relokalisiert die einzelnen Programmteile

```
2663-  86 10       STX   $10            ; Aufruf mit Startadresse der Parameterliste
2665-  84 11       STY   $11              in X(lo) und Y(hi)
2667-  A0 00       LDY   #$00
2669-  B1 10       LDA   ($10),Y
266B-  C9 05       CMP   #$05
266D-  B0 72       BCS   $26E1          ; $FF: Listenende
266F-  AA          TAX                  ; „Type"
2670-  C8          INY
2671-  B1 10       LDA   ($10),Y
2673-  85 14       STA   $14
2675-  C8          INY                  ; $0014/15: Zieladresse
2676-  B1 10       LDA   ($10),Y
2678-  85 15       STA   $15
267A-  C8          INY
267B-  B1 10       LDA   ($10),Y
267D-  85 16       STA   $16
267F-  C8          INY                  ; $0016/17: Anzahl der Bytes
2680-  B1 10       LDA   ($10),Y
2682-  85 17       STA   $17
2684-  30 5D       BMI   $26E3
2686-  8A          TXA
2687-  F0 63       BEQ   $26EC          ; „Type" 0: Bereich löschen
2689-  C8          INY
268A-  B1 10       LDA   ($10),Y        ; sonst weitere Parameter
268C-  85 12       STA   $12
268E-  85 18       STA   $18
2690-  C8          INY                  ; $0012/13: Quelladresse
2691-  18          CLC
2692-  65 16       ADC   $16
2694-  85 1A       STA   $1A
```

```
2696-  B1 10      LDA   ($10),Y
2698-  85 13      STA   $13
269A-  85 19      STA   $19          ; $0018/19: Kopie von $0012/13
269C-  65 17      ADC   $17
269E-  85 1B      STA   $1B          ; $001A/1B: Obergrenze
26A0-  CA         DEX
26A1-  F0 6D      BEQ   $2710        ; „Type" 1: Kopie von „Quelle" nach „Ziel"

26A3-  8E 3C 28   STX   $283C        ; „Typen" 2/3 (Re-Set jedes 1./2. Bytes)
26A6-  C8         INY                 und 4 (Disassembly und Re-Set absoluter
26A7-  B1 10      LDA   ($10),Y       Referenzen
26A9-  8D 3D 28   STA   $283D        ; Anzahl(!) der Referenzbereiche
26AC-  AA         TAX
26AD-  C8         INY
26AE-  B1 10      LDA   ($10),Y
26B0-  9D 3E 28   STA   $283E,X      ; Untergrenzen der Referenzbereiche
26B3-  CA         DEX
26B4-  10 F7      BPL   $26AD
26B6-  AE 3D 28   LDX   $283D
26B9-  C8         INY
26BA-  B1 10      LDA   ($10),Y
26BC-  9D 46 28   STA   $2846,X      ; Obergrenzen der Referenzbereiche
26BF-  CA         DEX
26C0-  10 F7      BPL   $26B9
26C2-  AE 3D 28   LDX   $283D
26C5-  C8         INY
26C6-  B1 10      LDA   ($10),Y
26C8-  9D 4E 28   STA   $284E,X      ; Additionsfaktoren für den Fall, daß
26CB-  CA         DEX                 die Referenz im Bereich liegt. Subtraktion
26CC-  10 F7      BPL   $26C5         erfolgt durch Overflow! (8-Bit Addition)
26CE-  20 16 27   JSR   $2716        ; erhöht den Parameter-Pointer
26D1-  AE 3C 28   LDX   $283C
26D4-  E0 03      CPX   #$03         ; war das „Type 4"?
26D6-  F0 0E      BEQ   $26E6
26D8-  20 7A 27   JSR   $277A        ; ansonsten „Type 3", setzt jedes 2. Byte
26DB-  20 23 27   JSR   $2723        ; danach Umkopieren – nicht benutzt
26DE-  4C 67 26   JMP   $2667        ; nächster Parametersatz in der Liste
26E1-  18         CLC
26E2-  60         RTS                ; Exit bei Ende der Parameterliste
26E3-  4C B0 27   JMP   $27B0
26E6-  20 8C 27   JSR   $278C        ; „Type 4" Handler: Disasm und Re-Set
26E9-  4C DB 26   JMP   $26DB        ; danach Umkopieren – nicht genutzt
```

– Type 0: Löschen eines Bereichs

```
26EC-  20 16 27   JSR   $2716        ; setzt Pointer auf nächsten Parametersatz
26EF-  A9 00      LDA   #$00
26F1-  A4 17      LDY   $17          ; $0016/17: Anzahl der Bytes
26F3-  F0 0C      BEQ   $2701        ; keine ganzen Seiten mehr übrig
26F5-  A8         TAY
26F6-  91 14      STA   ($14),Y      ; $0014/15: Zieladresse
26F8-  C8         INY
26F9-  D0 FB      BNE   $26F6        ; Loop für $100 Bytes
26FB-  E6 15      INC   $15          ; Zieladresse + $100
26FD-  C6 17      DEC   $17          ; Seitenzähler minus 1
26FF-  D0 F5      BNE   $26F6
2701-  A4 16      LDY   $16
2703-  F0 08      BEQ   $270D        ; keine einzelnen Bytes mehr übrig
2705-  98         TYA
2706-  91 14      STA   ($14),Y
2708-  C8         INY
2709-  C4 16      CPY   $16          ; Vergleich mit Anzahl – Lobyte
270B-  90 F9      BCC   $2706
270D-  4C 67 26   JMP   $2667        ; nächster Parametersatz aus der Liste
```

– Type 1 „Handler"

```
2710-  20 16 27   JSR   $2716        ; Parameter-Pointer auf nächsten Satz
2713-  4C DB 26   JMP   $26DB        ; ruft dann die Kopier-Routine auf
```

– Setzt den Pointer in der Parameterliste auf den nächsten Satz

```
2716-  98         TYA                ; Zähler und momentaner Index, steht
2717-  38         SEC                  im Moment auf dem letzten bereits
2718-  65 10      ADC   $10            erfaßten Parameter, deshalb Addition
271A-  85 10      STA   $10            mit SEC (+1).
271C-  A5 11      LDA   $11          ; $0010/11: Pointer zum Parametersatz
271E-  69 00      ADC   #$00
2720-  85 11      STA   $11
2722-  60         RTS
```

– Type 1: Kopier-Routine

```
2723-  A5 13      LDA   $13          ; $0012/13: Quelle
2725-  C5 15      CMP   $15          ; $0014/15: Ziel
2727-  90 0B      BCC   $2734
2729-  D0 2C      BNE   $2757          wenn Zieladresse < Quelladresse,
272B-  A5 12      LDA   $12            wird in aufsteigender Reihenfolge kopiert
272D-  C5 14      CMP   $14
272F-  90 03      BCC   $2734
2731-  D0 24      BNE   $2757
2733-  60         RTS                ; RTS, wenn Quelle und Ziel identisch sind

2734-  A4 17      LDY   $17          ; Ziel > Quelle:
2736-  98         TYA
2737-  18         CLC
2738-  65 13      ADC   $13          ; Kopie in absteigender Reihenfolge:
273A-  85 13      STA   $13            Zuerst werden beide Pointer um die
273C-  98         TYA                  Anzahl der zu kopierenden Speicherseiten
273D-  18         CLC                  erhöht.
273E-  65 15      ADC   $15
2740-  85 15      STA   $15
2742-  A4 16      LDY   $16          ; danach Start mit einzelnen Bytes
2744-  F0 08      BEQ   $274E        ; wenn auf Seitengrenze, DEC des Zählers(hi)
2746-  88         DEY
2747-  B1 12      LDA   ($12),Y      ; Kopieren einzelner Bytes
2749-  91 14      STA   ($14),Y
274B-  98         TYA
274C-  D0 F8      BNE   $2746
274E-  C6 15      DEC   $15          ; danach werden die ganzen Seiten kopiert
2750-  C6 13      DEC   $13
2752-  C6 17      DEC   $17          ; Herunterzähler für ganze Seiten
2754-  10 F0      BPL   $2746
2756-  60         RTS

2757-  A0 00      LDY   #$00         ; Zieladresse < Quelle: aufsteigende Folge
2759-  A5 17      LDA   $17          ; Sprung, wenn keine ganzen Seiten
275B-  F0 0F      BEQ   $276C          zu kopieren sind
275D-  B1 12      LDA   ($12),Y
275F-  91 14      STA   ($14),Y      ; sonst die ganzen Seiten zuerst
2761-  C8         INY
2762-  D0 F9      BNE   $275D
2764-  E6 15      INC   $15
2766-  E6 13      INC   $13
2768-  C6 17      DEC   $17          ; Herunterzähler für ganze Seiten
276A-  D0 F1      BNE   $275D
276C-  A5 16      LDA   $16          ; einzelne Bytes übrig?
276E-  F0 09      BEQ   $2779
```

```
2770-  B1 12      LDA  ($12),Y      ; Kopie einzelner Bytes
2772-  91 14      STA  ($14),Y
2774-  C8         INY
2775-  C4 16      CPY  $16          ; Vergleich mit Byte-Anzahl (lo)
2777-  D0 F7      BNE  $2770
2779-  60         RTS
```

– TYPE 2/3: Re-Set jedes Nten Bytes. N = Type-1 (=> $283C)

```
277A-  AC 3C 28   LDY  $283C        ; N
277D-  88         DEY
277E-  B1 18      LDA  ($18),Y      ; Pointer auf den Code
2780-  20 B8 27   JSR  $27B8        ; Test auf Bereich(e) und ggf. Re-Set
2783-  AD 3C 28   LDA  $283C        ; N
2786-  20 D4 27   JSR  $27D4        ; $0018/19 plus N, Vergleich mit Obergrenze
2789-  90 EF      BCC  $277A        ; Loop bis zur Obergrenze
278B-  60         RTS
```

– TYPE 4: Disassembly und Korrektur absoluter Referenzen

```
278C-  A0 00      LDY  #$00
278E-  B1 18      LDA  ($18),Y      ; Pointer auf den Code
2790-  20 E7 27   JSR  $27E7        ; berechnet die Länge einer Instruktion
2793-  F0 11      BEQ  $27A6        ; ERROR: Ungültiger Opcode
2795-  C9 03      CMP  #$03         ; 3-Byte-Instruktion, d.h. absolute Ref?
2797-  D0 07      BNE  $27A0        ; nein
2799-  A0 02      LDY  #$02         ; das 2. Byte wird geprüft
279B-  20 B8 27   JSR  $27B8          und ggf. korrigiert
279E-  A9 03      LDA  #$03         ; noch einmal: Länge der Instruktion
27A0-  20 D4 27   JSR  $27D4        ; $0018/19 + Instr.Länge, Test auf Obergrenze
27A3-  90 E7      BCC  $278C        ; Loop, wenn Obergrenze noch nicht erreicht
27A5-  60         RTS

27A6-  68         PLA               ; ERR EXIT – einzige Fehlermöglichkeit im
27A7-  68         PLA                 MOVER – ungültiger Opcode bei Type 4
27A8-  A6 18      LDX  $18          ; diese Daten werden nicht verwertet,
27AA-  A4 19      LDY  $19
27AC-  A9 FF      LDA  #$FF         ; dieser Wert auch nicht
27AE-  38         SEC               ; RTS zur Routine, die den MOVER aufgerufen
27AF-  60         RTS                 hat => „RELOCATION/CONFIGURATION ERROR"

27B0-  A6 10      LDX  $10          ; nicht benutzt und wohl von einer
27B2-  A4 11      LDY  $11            früheren Version stehengeblieben
27B4-  A9 00      LDA  #$00           (s. „Spezielles")
27B6-  38         SEC
27B7-  60         RTS
```

– Korrektur des n.ten Bytes, falls im Referenz-Zielbereich

```
27B8-  B1 18      LDA  ($18),Y      ; Aufruf mit n in Y-Reg
27BA-  AE 3D 28   LDX  $283D        ; kann > 1 sein!
27BD-  DD 3E 28   CMP  $283E,X      ; Untergrenze
27C0-  90 07      BCC  $27C9        ; außerhalb des Referenz-Zielbereichs
27C2-  DD 46 28   CMP  $2846,X      ; Obergrenze
27C5-  90 06      BCC  $27CD        ; ok. Innerhalb des Zielbereichs
27C7-  F0 04      BEQ  $27CD
27C9-  CA         DEX
27CA-  10 F1      BPL  $27BD        ; Loop durch alle gesetzten Bereiche
27CC-  60         RTS
27CD-  18         CLC
27CE-  7D 4E 28   ADC  $284E,X      ; Addition (mit Overflow: Subtraktion) des
27D1-  91 18      STA  ($18),Y        Korrekturfaktors für diesen Zielbereich
27D3-  60         RTS
```

– Erhöht $0018/19 und testet gegen Obergrenze

```
27D4-  18          CLC                 ; Aufruf mit Instr.Länge bzw. Schritt-
27D5-  65 18       ADC   $18             weite im Akku
27D7-  A4 19       LDY   $19
27D9-  90 01       BCC   $27DC
27DB-  C8          INY                 ; Erhöhung des Zielpointers (hi)
27DC-  C4 1B       CPY   $1B
27DE-  90 02       BCC   $27E2
27E0-  C5 1A       CMP   $1A           ; $001A/1B: Obergrenze
27E2-  85 18       STA   $18
27E4-  84 19       STY   $19           ; neue Adresse wird immer gesetzt
27E6-  60          RTS                 ; mit SEC, wenn Obergrenze ereicht
```

– Bestimmung der Länge einer Instruktion

```
27E7-  48          PHA                 ; Aufruf mit dem ersten Instruktions-Byte
27E8-  29 03       AND   #$03            im Akku
27EA-  A8          TAY                 ; Bereich: 00-03
27EB-  68          PLA
27EC-  4A          LSR
27ED-  4A          LSR
27EE-  AA          TAX
27EF-  BD FC 27    LDA   $27FC,X       ; Tabellenbereich: $00-$3F
27F2-  88          DEY
27F3-  30 04       BMI   $27F9
27F5-  4A          LSR
27F6-  4A          LSR
27F7-  D0 F9       BNE   $27F2
27F9-  29 03       AND   #$03          ; $00: ungültiger Opcode, sonst Länge
27FB-  60          RTS
```

– Tabelle für Instruktionslängen

```
27FC- 09 28 19 3C 0A 28 0D 3C  0B 2A 19 3F 0A 28 0D 3C
280C- 09 28 19 3F 0A 28 0D 3C  09 28 19 3F 0A 28 0D 3C
281C- 08 2A 11 3F 0A 2A 1D 0C  2A 2A 19 3F 0A 2A 1D 3F
282C- 0A 2A 19 3F 0A 28 0D 3C  0A 2A 19 3F 0A 28 0D 3C

283C- 00                       ; N (Schrittweite) für Typen 2/3 (Relocate)
283D- 00                       ; Anzahl der Referenzbereiche (Relocate)
283E- 00 00 00 00 00 00 00 00  ; Platz für Untergrenzen
2846- 00 00 00 00 00 00 00 00  ; Platz für Obergrenzen
284E- 00 00 00 00 00 00 00 00  ; Korrekturwerte - selber Index wie Bereichsgrenzen

2856- 00 00 00 00 00 00 00 00  00 00                    ..........
2860- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
2870- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
2880- 00 00 00                                 00 00 00 00 00  ................
2890- 00 00 00  anscheinend unbenutzt          00 00 00 00 00  ................
28A0- 00 00 00                                 00 00 00 00 00  ................
28B0- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
28C0- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
28D0- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
28E0- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
28F0- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00     ...............
```

## – Installation der RAMDisk-RWTB

```
28FF-   EA          NOP
2900-   A0 99       LDY   #$99          ; das Interface zur RAMDisk-RWTB wird von
2902-   B9 00 2C    LDA   $2C00,Y         $2C00 nach $FF00 (Bank 1) kopiert
2905-   99 00 FF    STA   $FF00,Y
2908-   88          DEY
2909-   C0 FF       CPY   #$FF
290B-   D0 F5       BNE   $2902
290D-   A2 00       LDX   #$00
290F-   86 3C       STX   $3C
2911-   CA          DEX
2912-   86 3E       STX   $3E
2914-   A2 2A       LDX   #$2A
2916-   86 3D       STX   $3D           ; Setup für Block-Transport:
2918-   E8          INX
2919-   86 3F       STX   $3F             $003C/3D: $2A00 Startadresse
291B-   A9 00       LDA   #$00
291D-   85 42       STA   $42             $003E/3F: $2BFF Endadresse
291F-   A9 02       LDA   #$02
2921-   85 43       STA   $43             $0042/43: $0200 Zieladresse
2923-   38          SEC                 ; von „unten" nach „oben"
2924-   20 11 C3    JSR   $C311         ; => JMP MOVEAUX (Crossbank Memory Move)
2927-   A9 00       LDA   #$00
2929-   8D 26 BF    STA   $BF26
292C-   A9 FF       LDA   #$FF          ; RWTB-Start für Slot 3 auf $FF00 in der
292E-   8D 27 BF    STA   $BF27           Startadressen-Tabelle der Global Page
2931-   EE 31 BF    INC   $BF31         ; DEVNUM + 1
2934-   AE 31 BF    LDX   $BF31         ; die RAM Disk wird als oberstes (erstes)
2937-   A9 BF       LDA   #$BF            Unit in die DEVTBL eingetragen
2939-   9D 32 BF    STA   $BF32,X       ; DRIVE 2 / SLOT 3 / RAM DISK
293C-   60          RTS

293D..29FF      ; unbenutzt

2A00..2BFF      ; RAMDisk-RWTB, nach Aux $0200..03FF kopiert

2C00..2C69      ; Interface für die RAM Disk - auf $FF00-$FF99 kopiert, nicht verschoben

2D00..4DFF      ; MLI: auf $D000.. verschoben und kopiert

4E00..4EFF      ; System Global Page für 64/128k

4F00..4FFF      ; System Global Page für 48k

5000..507D      ; SET TIME f.Thunderclock - auf $F142 (48k: $B142) versch. + kopiert

5080..50FF      ; IRQ/RESET Handler - auf $F800 kopiert (nur 64/128k)

5100..51FF      ; unbenutzt (Nuller)

5200..58FF      ; RWTB für DISKII - auf $F800 kopiert
                  (48k: auf $B800 kopiert + verschoben)

5900..61FF      ; REBOOT - auf $D100 in der Bank 2 kopiert (nur 64/128k)
```

# 2.3. REBOOT

**Original-Lage:** $D100 - $D3FF in der Bank 2
**Aufruf:** MLI Call mit COMMAND $65 und vier(!) Parametern

Nach Überprüfung des COMMAND-Byte mit der Tabelle auf $EF25 wird die Anzahl der COMMAND-Parameter überprüft. REBOOT muß 4 Parameter haben, die Parameter selber spielen dabei keine Rolle (Dummies).
Die Kopier-Routine auf $EECF kopiert REBOOT aus der Bank 2 der Language Card von $D100-$D4FF nach $1000-$13FF. Die dabei benutzten Pointer ($00-$03) werden zwischengespeichert, im weiteren Verlauf von REBOOT allerdings nicht mehr benutzt. Der RESET-Vektor wird auf $1000 gesetzt. Danach Sprung zu $1000, dem Start des eigentlichen REBOOT.
REBOOT ist offensichtlich zu einem späteren Zeitpunkt in ProDOS eingebaut worden. Vergleicht man den Stil der Programmierung mit dem restlichen MLI, wird offensichtlich, daß der Programmierer von ProDOS es hier entweder schrecklich eilig gehabt hat, oder – wesentlich wahrscheinlicher – daß dieser Programmteil von jemand anderem erstellt wurde.

```
1000-   AD 82 C0    LDA   $C082         ; ROMON
1003-   20 93 FE    JSR   $FE93         ; SETVID (PR# 0)
1006-   20 89 FE    JSR   $FE89         ; SETKBD (IN #0)
1009-   8D 0C C0    STA   $C00C
100C-   8D 0F C0    STA   $C00F         ; extended 80-Zeichen aus
100F-   8D 00 C0    STA   $C000
1012-   A2 17       LDX   #$17
1014-   A9 01       LDA   #$01          ; Neubelegung der System Bit Map:
1016-   9D 58 BF    STA   $BF58,X         die Speicherseite $BF00 wird belegt,
1019-   CA          DEX
101A-   A9 00       LDA   #$00
101C-   9D 58 BF    STA   $BF58,X       ; alle anderen Seiten werden freigegeben,
101F-   CA          DEX
1020-   10 FA       BPL   $101C
1022-   A9 CF       LDA   #$CF          ; 1100 1111 - Seiten $00/$01 belegt,
1024-   8D 58 BF    STA   $BF58           $02/$03 frei, $04..$07 belegt (!)
```

**– GET SYSTEM PREFIX**

```
1027-   20 58 FC    JSR   $FC58         ; HOME
102A-   20 8E FD    JSR   $FD8E         ; CROUT
102D-   A9 41       LDA   #$41
102F-   8D E9 11    STA   $11E9
1032-   A9 12       LDA   #$12          ; Startadresse des 1. Texts: $1241
1034-   8D EA 11    STA   $11EA
1037-   20 E6 11    JSR   $11E6         ; „ENTER PREFIX (PRESS RETURN...."
103A-   A9 03       LDA   #$03
103C-   85 25       STA   $25           ; Zeile 4 im TEXT-Bildschirm
103E-   20 8E FD    JSR   $FD8E         ; CROUT (benutzt VCALC)
```

```
1041-  20 00 BF   JSR   $BF00        ; GET PREFIX
1044-  C7 F8 12                     ; Parmblock: $12F8, Zieladresse $0280
1047-  AE 80 02   LDX   $0280        ; Längenbyte
104A-  A9 00      LDA   #$00
104C-  9D 81 02   STA   $0281,X      ; 00 = Endmarker für Print-Routine
104F-  A9 81      LDA   #$81
1051-  8D E9 11   STA   $11E9
1054-  A9 02      LDA   #$02         ; Startadresse für Print-Routine: $0281
1056-  8D EA 11   STA   $11EA
1059-  20 E6 11   JSR   $11E6        ; druckt das PREFIX (oder auch nichts)
105C-  A2 00      LDX   #$00
105E-  C6 25      DEC   $25          ; CROUT - Cursor bleibt auf derselben Zeile,
1060-  20 8E FD   JSR   $FD8E          d.h. geht auf das erste Zeichen d. Prefix
```

– **GET PREFIX** vom Benutzer

```
1063-  20 0C FD   JSR   $FD0C        ; RDKEY - erstes (nächstes) Zeichen
1066-  C9 8D      CMP   #$8D         ; <CR>?
1068-  F0 4E      BEQ   $10B8        ; PREFIX akzeptiert oder Eingabe-Ende

106A-  48         PHA                ; anderes Zeichen (nicht <CR>)
106B-  20 9C FC   JSR   $FC9C        ; clear end-of-line (Monitor) bzw. Zeile
106E-  68         PLA                      ab Cursor, wenn Backspace
106F-  C9 9B      CMP   #$9B         ; <ESC>?
1071-  F0 B4      BEQ   $1027        ; Ja, Restart (GET PREFIX)
1073-  C9 98      CMP   #$98         ; ctrl-X?
1075-  F0 B0      BEQ   $1027        ; Ja, Restart
1077-  C9 89      CMP   #$89         ; ↑I ist nicht erlaubt:
1079-  F0 13      BEQ   $108E        ; BELL, danach nächstes Zeichen
107B-  C9 88      CMP   #$88
107D-  D0 0D      BNE   $108C
107F-  E0 00      CPX   #$00         ; BACKSPACE
1081-  F0 03      BEQ   $1086        ; wenn nicht erstes Zeichen => CLEOL, um
1083-  C6 24      DEC   $24            das Zeichen zu löschen
1085-  CA         DEX
1086-  20 9C FC   JSR   $FC9C
1089-  4C 63 10   JMP   $1063        ; nächstes Zeichen

108C-  B0 06      BCS   $1094        ; Zeichen ist nicht BS/ESC/↑X/↑I/CR
108E-  20 3A FF   JSR   $FF3A        ; wenn < BS, BELL
1091-  4C 63 10   JMP   $1063        ; nächstes Zeichen
```

– Übersetzung in Großbuchstaben und Test auf illegale Zeichen

```
1094-  C9 DB      CMP   #$DB         ;  <„Ä" ? (d.h. Großbuchstabe)?
1096-  90 02      BCC   $109A
1098-  29 DF      AND   #$DF         ; sonst Übersetzung
109A-  C9 AE      CMP   #$AE
109C-  90 F0      BCC   $108E        ; < „."?
109E-  C9 DB      CMP   #$DB
10A0-  B0 EC      BCS   $108E        ; ERR für Zeichen > „Ä" nach Übersetzung
10A2-  C9 BA      CMP   #$BA
10A4-  90 04      BCC   $10AA
10A6-  C9 C1      CMP   #$C1
10A8-  90 E4      BCC   $108E        ; (;<>=?@) sind nicht erlaubt
10AA-  E8         INX
10AB-  E0 27      CPX   #$27         ; Zeichen ok. Namenslänge jetzt?
10AD-  B0 C6      BCS   $1075        ; wenn max. erreicht, keine weiteren Zeichen
10AF-  9D 80 02   STA   $0280,X      ; ok. Speichern
10B2-  20 ED FD   JSR   $FDED        ; PRINT
10B5-  4C 63 10   JMP   $1063        ; nächstes Zeichen – Loop bis <CR>
```

## – GOT PREFIX NAME

```
10B8-   E0 00       CPX   #$00        ; Namenslänge = 00?
10BA-   F0 12       BEQ   $10CE        ; ist ok. - kein PREFIX
10BC-   8E 80 02    STX   $0280        ; Setzen der Namenslänge
10BF-   20 00 BF    JSR   $BF00        ; SET PREFIX
10C2-   C6 F8 12                       ; Parmblock $12F8, Start des Namens $0280
10C5-   90 07       BCC   $10CE
10C7-   20 3A FF    JSR   $FF3A        ; jeder MLI-Fehler: BELL
10CA-   A9 00       LDA   #$00
10CC-   F0 A7       BEQ   $1075        ; Restart
```

## – PREFIX SET

```
10CE-   20 58 FC    JSR   $FC58        ; HOME
10D1-   20 8E FD    JSR   $FD8E        ; CROUT
10D4-   A9 69       LDA   #$69
10D6-   8D E9 11    STA   $11E9
10D9-   A9 12       LDA   #$12         ; Start des nächsten Texts: $1269
10DB-   8D EA 11    STA   $11EA
10DE-   20 E6 11    JSR   $11E6        ; „ENTER PATHNAME OF NEXT APPLICATION"
10E1-   A9 03       LDA   #$03
10E3-   85 25       STA   $25
10E5-   20 8E FD    JSR   $FD8E        ; line #4
10E8-   A2 00       LDX   #$00
```

## – GET PROGRAM NAME

```
10EA-   A9 FF       LDA   #$FF         ; „Prompt"
10EC-   20 ED FD    JSR   $FDED        ; PRINT
10EF-   C6 24       DEC   $24          ; Cursor zurück auf „Prompt"
10F1-   AD 00 C0    LDA   $C000        ; Warten auf ein Zeichen vom Keyboard
10F4-   10 FB       BPL   $10F1
10F6-   8D 10 C0    STA   $C010
10F9-   C9 9B       CMP   #$9B         ; <ESC>?
10FB-   D0 06       BNE   $1103
10FD-   A5 24       LDA   $24          ; wenn bereits Zeichen im Programm-Namen,
10FF-   D0 CD       BNE   $10CE          dann Restart bei PREFIX SET
1101-   F0 C9       BEQ   $10CC        ; ansonsten Restart bei GET SYSTEM PREFIX
1103-   C9 98       CMP   #$98         ; ctrl-X?
1105-   F0 C7       BEQ   $10CE        ; ja, Restart bei PREFIX SET
1107-   C9 89       CMP   #$89
1109-   F0 09       BEQ   $1114        ; BELL für ctrl-I
110B-   C9 88       CMP   #$88
110D-   D0 03       BNE   $1112
110F-   4C D0 11    JMP   $11D0        ; BS-Patch. „Returnt" zu $10EA. Aua...
1112-   B0 06       BCS   $111A
1114-   20 3A FF    JSR   $FF3A        ; char < BS. BELL
1117-   4C EA 10    JMP   $10EA        ; und nächstes Zeichen
111A-   C9 8D       CMP   #$8D         ; <CR>?
111C-   F0 29       BEQ   $1147        ; ok. Eingabe beendet
111E-   C9 DB       CMP   #$DB
1120-   90 02       BCC   $1124
1122-   29 DF       AND   #$DF         ; Großbuchstaben-Übersetzung, s. GET PREFIX
1124-   C9 AE       CMP   #$AE
1126-   90 EC       BCC   $1114
1128-   C9 DB       CMP   #$DB
112A-   B0 E8       BCS   $1114
112C-   C9 BA       CMP   #$BA
112E-   90 04       BCC   $1134
1130-   C9 C1       CMP   #$C1
1132-   90 E0       BCC   $1114
1134-   48          PHA
```

```
1135-  20 9C FC    JSR   $FC9C        ; Zeichen gültig. CLEOL, um das „Prompt"
1138-  68          PLA                   zu löschen,
1139-  20 ED FD    JSR   $FDED        ; PRINT des neuen Zeichens
113C-  E8          INX
113D-  E0 27       CPX   #$27         ; max. Namenslänge erreicht?
113F-  B0 C4       BCS   $1105
1141-  9D 80 02    STA   $0280,X      ; wenn nicht überschritten, Speichern
1144-  4C EA 10    JMP   $10EA        ; Loop bis <CR>
```

## – GOT NAME OF PROGRAM

```
1147-  A9 A0       LDA   #$A0
1149-  20 ED FD    JSR   $FDED        ; „Prompt" mit einem Leerzeichen löschen
114C-  8E 80 02    STX   $0280        ; Namenslänge speichern
114F-  20 00 BF    JSR   $BF00
1152-  C4 D1 12                       ; GET F'INFO, Parmblock $12D1, Name $0280
1155-  90 03       BCC   $115A
1157-  4C F6 11    JMP   $11F6        ; MLI-Fehler: Error Handler
115A-  AD D5 12    LDA   $12D5
115D-  C9 FF       CMP   #$FF         ; File Type(!) von GET FILE INFO
115F-  F0 05       BEQ   $1166        ; ok
1161-  A9 01       LDA   #$01         ; Index zu NOT A TYPE „SYS" FILE
1163-  4C F6 11    JMP   $11F6        ; Error Handler
1166-  A9 00       LDA   #$00
1168-  8D EA 12    STA   $12EA        ; Refno für folgendes CLOSE (= ALL)
116B-  20 00 BF    JSR   $BF00
116E-  CC E9 12                       ; CLOSE ALL, Parmblock $12E9
1171-  90 03       BCC   $1176
1173-  4C F6 11    JMP   $11F6        ; MLI-Fehler: Error Handler
1176-  AD D4 12    LDA   $12D4
1179-  29 01       AND   #$01         ; ACCESS von GET F'INFO
117B-  D0 05       BNE   $1182        ; READ erlaubt?
117D-  A9 27       LDA   #$27
117F-  4C F6 11    JMP   $11F6        ; Error Handler: „I/O ERROR"
1182-  20 00 BF    JSR   $BF00
1185-  C8 E3 12                       ; OPEN File, Parmblock $12E3, Filebuffer $1800
1188-  90 03       BCC   $118D
118A-  4C F6 11    JMP   $11F6        ; ERR beim OPEN
118D-  AD E8 12    LDA   $12E8        ; zurückgebrachte Reference Number
1190-  8D EC 12    STA   $12EC        ; Parmblock für GET EOF
1193-  8D F4 12    STA   $12F4        ; Parmblock für READ
1196-  20 00 BF    JSR   $BF00
1199-  D1 F3 12                       ; GET EOF, Parmblock $12F3
119C-  90 03       BCC   $11A1
119E-  4C F6 11    JMP   $11F6        ; ERR bei GET EOF
11A1-  AD F7 12    LDA   $12F7        ; MSByte der File-Länge
11A4-  F0 05       BEQ   $11AB
11A6-  A9 27       LDA   #$27         ; (System program too long)
11A8-  4C F6 11    JMP   $11F6
11AB-  AD F5 12    LDA   $12F5        ; Setzen des Request Count für READ
11AE-  8D EF 12    STA   $12EF           mit den Daten von GET EOF
11B1-  AD F6 12    LDA   $12F6
11B4-  8D F0 12    STA   $12F0
11B7-  20 00 BF    JSR   $BF00
11BA-  CA EB 12                       ; READ, Parmblock $12EB, Zielbereich: $2000...
11BD-  08          PHP
11BE-  20 00 BF    JSR   $BF00
11C1-  CC E9 12                       ; CLOSE ALL
11C4-  90 04       BCC   $11CA
11C6-  28          PLP                ; ERR bei CLOSE
11C7-  4C F6 11    JMP   $11F6
11CA-  28          PLP                ; irgendein Fehler bei READ?
```

```
11CB-   B0 FA       BCS   $11C7
11CD-   4C 00 20    JMP   $2000        ; Nein, Start des geladenen Programms
```

**– BS „Patch":** Löschen des letzten Zeichens und des Prompts im Bildschirm

```
11D0-   A5 24       LDA   $24          ; Cursor horizontal (CH)
11D2-   F0 0F       BEQ   $11E3        ; wenn ganz links in der Zeile, „Return"
11D4-   CA          DEX
11D5-   A9 A0       LDA   #$A0
11D7-   20 ED FD    JSR   $FDED        ; sonst Überschreiben des Prompts mit
11DA-   C6 24       DEC   $24                  Leerzeichen
11DC-   C6 24       DEC   $24
11DE-   20 ED FD    JSR   $FDED        ; und „Blanken" des letzten Zeichens
11E1-   C6 24       DEC   $24
11E3-   4C EA 10    JMP   $10EA        ; „Return": nächstes Zeichen im Filenamen
```

**– PRINT MESSAGE**

```
11E6-   A2 00       LDX   #$00
11E8-   BD ** **    LDA   $11E8,X      ; Basis-Adresse durch Caller gesetzt
11EB-   F0 08       BEQ   $11F5        ; 00 = Endmarker
11ED-   09 80       ORA   #$80
11EF-   20 ED FD    JSR   $FDED        ; COUT (PRINT) vom Monitor
11F2-   E8          INX
11F3-   D0 F3       BNE   $11E8
11F5-   60          RTS
```

**– ERR HANDLER**

```
11F6-   85 DE       STA   $DE          ; ERRNO vom MLI oder von REBOOT gesetzt
11F8-   A9 0C       LDA   #$0C
11FA-   85 25       STA   $25
11FC-   20 8E FD    JSR   $FD8E        ; line #13
11FF-   A5 DE       LDA   $DE
1201-   C9 01       CMP   #$01
1203-   D0 0C       BNE   $1211
1205-   A9 8C       LDA   #$8C
1207-   8D E9 11    STA   $11E9
120A-   A9 12       LDA   #$12         ; ERR #1 (von REBOOT gesetzt):
120C-   8D EA 11    STA   $11EA
120F-   D0 26       BNE   $1237        ; NOT A TYPE „SYS" FILE
1211-   C9 40       CMP   #$40
1213-   F0 18       BEQ   $122D
1215-   C9 44       CMP   #$44         ; ERRs $40/$44/$45/$46 (vom MLI):
1217-   F0 14       BEQ   $122D
1219-   C9 45       CMP   #$45         ; (INVALID PATHNAME oder
121B-   F0 10       BEQ   $122D
121D-   C9 46       CMP   #$46         ; VOL/DIR/FILE NOT FOUND)
121F-   F0 0C       BEQ   $122D
1221-   A9 A3       LDA   #$A3
1223-   8D E9 11    STA   $11E9        ; keiner dieser Fehler:
1226-   A9 12       LDA   #$12
1228-   8D EA 11    STA   $11EA        ; => „I/O ERROR"
122B-   D0 0A       BNE   $1237
122D-   A9 BA       LDA   #$BA
122F-   8D E9 11    STA   $11E9        ; Sprung hierher für ERR $40/$44/$45/$46
1232-   A9 12       LDA   #$12
1234-   8D EA 11    STA   $11EA        ; FILE/PATH NOT FOUND

1237-   20 E6 11    JSR   $11E6        ; PRINT der Error Message
123A-   A9 00       LDA   #$00
```

```
123C-   85 25       STA   $25
123E-   4C D1 10    JMP   $10D1        ; Restart von REBOOT mit PREFIX SET (!)
```

– Prompt lines und Error Messages. 00 = Textende

```
1241- C5 CE D4 C5 D2 A0 D0 D2  C5 C6 C9 D8 A0 A8 D0 D2    ENTER PREFIX (PR
1251- C5 D3 D3 A0 A2 D2 C5 D4  D5 D2 CE A2 A0 D4 CF A0    ESS „RETURN" TO
1261- C1 C3 C3 C5 D0 D4 A9 00  C5 CE D4 C5 D2 A0 D0 C1    ACCEPT).ENTER PA
1271- D4 C8 CE C1 CD C5 A0 CF  C6 A0 CE C5 D8 D4 A0 C1    THNAME OF NEXT A
1281- D0 D0 CC C9 C3 C1 D4 C9  CF CE 00 87 CE CF D4 A0    PPLICATION..NOT
1291- C1 A0 D4 D9 D0 C5 A0 A2  D3 D9 D3 A2 A0 C6 C9 CC    A TYPE „SYS" FIL
12A1- C5 00 87 C9 AF CF A0 C5  D2 D2 CF D2 A0 A0 A0 A0    E..I/O ERROR
12B1- A0 A0 A0 A0 A0 A0 A0 A0  00 87 C6 C9 CC C5 AF D0            ..FILE/P
12C1- C1 D4 C8 A0 CE CF D4 A0  C6 CF D5 CE C4 A0 A0 00    ATH NOT FOUND  .
```

– Parameterblocks für MLI-Aufrufe

```
12D1- 0A 80 02 00 00 A0 A0 A0          ; GET F'INFO: Name $0280
12D9- A0 A0 A0 80 A0 A0 97 C3 A0 A0
12E3- 03 80 02 00 18 00                ; OPEN: Name $0280, Puffer $1800
12E9- 01 00                            ; CLOSE ALL
12EA- 04 ** 00 20 00 00 00 00          ; READ, Ziel-Start $2000, Req.Count von EOF
12F3- 02 ** A4 80 A0                   ; GET EOF, Reference Number von OPEN
12F8- 01 80 02                         ; GET/SET PREFIX, Start des Path $0280

12FB- 00 00 00 00 00                   ; unbenutzt
```

# Teil III – Andere Funktionen von ProDOS

# Interrupts/RESET und SET TIME für die Thunderclock

## 3.1. Die Verarbeitung von NMI, RESET und IRQ

Es gibt drei reguläre Möglichkeiten (d.h. Herausziehen des Chips oder Abschalten der Stromversorgung sind hier nicht berücksichtigt), den Prozessor des APPLE bei der Abarbeitung eines Programms zu unterbrechen.

- NMI (non maskable Interrupt)
- RESET
- IRQ (Interrupt-Request – maskierbarer Interrupt, auch durch BRK auszulösen)

Für jedes dieser Signale existiert eine eigene Prozessorleitung. Wird diese Leitung auf Nullpegel (low) gebracht, passiert folgendes:

- Der Prozessor bringt die gerade laufende Instruktion zu Ende.
- Liegt ein IRQ-Signal an, wird zuerst prozessorintern geprüft, ob das entsprechende Bit im Statusregister (P-Register) gelöscht ist, d.h. ob diese Art des Interrupts freigegeben (nicht maskiert) ist. Ist das nicht der Fall, geht die Abarbeitung des Programms bei einem IRQ-Signal völlig normal weiter, der IRQ wird ignoriert.
- Liegt ein nicht maskiertes IRQ-Signal oder ein NMI-Signal an, werden das Statusregister und der Programmzähler auf den Stack gebracht, bei einen RESET unterbleibt dieser Schritt.
- Der Prozessor liest ein 16-Bit-Wort von den folgenden Adressen und kopiert den gelesenen Wert in den Programmzähler, d.h. die Programmausführung geht ab der gelesenen Startadresse weiter:

| | |
|---|---|
| $FFFA/$FFFB für einen NMI | „JMP ($FFFA)" |
| $FFFC/$FFFD für RESET | „JMP ($FFFC)" |
| $FFFE/$FFFF für einen IRQ | „JMP ($FFFE)" |

Je nachdem, ob das Monitor-ROM oder die Language Card zum Zeitpunkt der Unterbrechung angeschaltet war, können die gelesenen Startadressen verschieden sein.

**1. Der NMI**

a) Im Monitor-ROM enthalten die Speicherstellen $FFFA/$FFFB die Startadresse $03FB. Auf $03FB wurde vom ProDOS-Boot kopiert:

```
03FB-   4C 59 FF    JMP  $FF59

FF59-   20 84 FE    JSR  $FE84      ; SETNORM (nicht INVERSE)
FF5C-   20 2F FB    JSR  $FB2F      ; INIT (setzt TEXT)
FF5F-   20 FE 93    JSR  $FE93      ; SETVID („PR#0")
FF62-   20 89 FE    JSR  $FE89      ; SETKBD („IN#0")
FF65-   D8          CLD
FF66-   20 3A FF    JSR  $FF3A      ; BELL
FF69-   D8          CLD
        .......
```

$FF69 ist die Startadresse des Monitors – hier „endet" der NMI-Handler, d.h. der Benutzer bekommt ein Sternchen zu sehen (ohne Anzeige der Registerinhalte) und kann sich via Control-C wieder in die BASIC-Ebene hineinretten. Im Klartext: Eine Behandlung des NMI ist im Monitor nicht vorgesehen.

b) In der Language Card (NMI während MLI-Ausführung) steht ebenfalls die Adresse $03FB auf den Speicherstellen $FFFA/$FFFB.
Da die Language Card durch einen NMI (genauso wie durch RESET oder IRQ) nicht ausgeschaltet wird, springt der Prozessor ebenfalls von $03FB nach $FF59 !
Ist das Interface zur RAMDisk-RWTB installiert, wird dort mitten hineingesprungen, ansonsten ist dieser Speicherbereich mehr oder weniger undefiniert. Die Folge ist ein Systemabsturz.

**2. RESET**

a) Im Monitor steht die Startadresse $FA62 auf den Speicherstellen $FFFC/$FFFD.
Der RESET-Ablauf ab $FA62 ist in den Kapiteln „Boot/Relocator" beschrieben. ProDOS setzt die Restart-Adresse ($03F2/$03F3 + „Powerup-Byte" $03F4) auf $FF59, d.h. auf den gleichen Einsprung wie den NMI-Handler. Ein RESET endet also ebenfalls im Monitor.
Anmerkung: Das BASIC.SYSTEM setzt den RESET-Vektor auf $03F2/$03F3 neu (auf $BE00).

b) Der RESET-Ablauf in der Language Card führt zu demselben Ergebnis (Sprung zu $FA62), vorher wird allerdings der Monitor wieder eingeschaltet. Aus diesem Grund steht hier auf den Speicherplätzen $FFFC/$FFFD die Startadresse $FFCB (immer noch innerhalb der Language Card):

```
        ......

FFC8-   8D 82 C0    STA   $C082        ; ROMON => „RTS" im Monitor
```

– RESET Handler

```
FFCB-   AD D7 FF    LDA   $FFD7        ; Monitor RESET - $FA62
FFCE-   48          PHA   $FA61        ; als „Return"-Adresse
FFCF-   AD D6 FF    LDA   $FFD6
FFD2-   48          PHA
FFD3-   4C C8 FF    JMP   $FFC8        ; Switch in den Monitor-ROM
FFD6-   61 FA                          ; diese RESET-Adresse wird gepusht
```

Auf $FFCB im Monitor steht ein „RTS". Folge: Der Prozessor „returnt" zu $FA62. Aufgrund einer etwas merkwürdigen Konstruktion des 6502 muß der Wert dieser Adresse minus 1 auf den Stack gebracht werden (die Returnadresse auf dem Stack wird bei der RTS-Instruktion noch einmal um eins erhöht).

### 3. Der IRQ

Dieses Signal wird – wie bereits gesagt – nur vom Prozessor berücksichtigt, wenn das entsprechende Flag im P-Register gelöscht ist (CLI-Instruktion).
a) Im Monitor steht hier die Startadresse $FA40 auf den Speicherstellen $FFFE/$FFFF.

```
FA40-   85 45       STA   $45         ; Speicherstelle ACC (Akku-Inhalt: REGDSP)
FA42-   68          PLA               ; P-Register!
FA43-   0A          ASL                 Bit 4 ist gesetzt, wenn der IRQ durch
FA44-   0A          ASL                 eine BRK-Instruktion (00) ausgelöst wurde
FA45-   0A          ASL
FA47-   30 03       BMI   $FA4C       ; BREAK mit Register-Display
FA49-   6C 03 FE    JMP   ($03FE)     ; $3FE: Startadresse des IRQ-Handlers
FA4C-   28          PLP
        .....
```

ProDOS setzt die Speicherstellen $03FE/$03FF auf den Wert $BFEB:

```
BFEB-   AD 8B C0    LDA   $C08B       ; BANKSELECT: Bank 1 (+ WRITE ENABLE)
BFEE-   AD 8B C0    LDA   $C08B
BFF1-   4C 3A D1    JMP   $D13A       ; IRQ-Handler im MLI
```

Die Routine auf $D13A ist zusammen mit dem restlichen MLI in den Kapiteln „Blockbeschreibung des MLI" und „Kommentiertes Listing des MLI" besprochen bzw. gelistet. Innerhalb dieser Routine werden der Reihe nach die Startadressen der möglichen Interrupt-Services geprüft und die Services angesprungen. Jeder Service muß einen Test enthalten, ob „seine" Quelle den IRQ ausgelöst hat.
Fällt dieser Test bei allen verzeichneten IRQ-Services negativ aus, d.h. kommt der IRQ aus unbekannter Quelle, folgt SYSTEM DEATH mit ERR#01.
Wird der Service erfolgreich ausgeführt, endet die Routine $D13A nach dem Zurückladen der vorher gespeicherten Register-Inhalte und einem Sprung zu $BFD0.

#### – INTEXIT

```
BFD0-   AD 8D BF    LDA   $BF8D       ; INTBANK1
BFD3-   F0 0D       BEQ   $BFE2       ; $00 = Bank 1 war (und ist immer noch) an
BFD5-   30 08       BMI   $BFDF       ; $FF = Bank 2 muß wieder angeschaltet werden
BFD7-   4A          LSR
BFD8-   90 0D       BCC   $BFE7       ; das ist ein „never", d.h. überflüssig
BFDA-   AD 82 C0    LDA   $C082       ; 01 = ROMON (IRQ während Monitor)
BFDD-   B0 08       BCS   $BFE7       ; „always"
```

```
BFDF-   AD 83 C0    LDA  $C083     ; Anschalten von Bank 2 (kein WRITE!)
BFE2-   A9 01       LDA  #$01
BFE4-   8D 8D BF    STA  $BF8D     ; und Setzen von „Monitor" f. nächsten IRQ
BFE7-   AD 88 BF    LDA  $BF88     ; Reload von INTAREG
BFEA-   40          RTI
```

Hier sind wohl einige Erklärungen nötig. Ein Teil der offenen Fragen beantwortet sich allerdings durch das folgende Listing.

b) In der Language Card enthalten die Speicherstellen $FFFE/$FFFF die Startadresse $FF9B (immer noch in der Language Card selber):

**– IRQ Handler**

```
FF9B-   48          PHA                ; Save des Akku
FF9C-   A5 45       LDA  $45           ; ACC – enthält im MLI RWTB-Pointer!
FF9E-   8D 56 BF    STA  $BF56         ; INTACCU
FFA1-   68          PLA                ; Akku-Inhalt zur Zeit des IRQ:
FFA2-   85 45       STA  $45             IRQ-Handler des Monitors erwartet
FFA4-   68          PLA                  diesen Wert in ACC
FFA5-   48          PHA                ; Status-Register (P-Reg)!
FFA6-   29 10       AND  #$10          ; IRQ durch BRK-Opcode?
FFA8-   D0 18       BNE  $FFC2         ; wenn ja, MON's BRK Routine
FFAA-   AD 00 D0    LDA  $D000         ; ansonsten – welche Bank ist an?
FFAD-   49 D8       EOR  #$D8          ; BANKID für Bank 1
FFAF-   F0 02       BEQ  $FFB3
FFB1-   A9 FF       LDA  #$FF
FFB3-   8D 8D BF    STA  $BF8D         ; INTBANK1: $00 (Bank 1) oder $FF (Bank 2)
FFB6-   8D 57 BF    STA  $BF57         ; $BF56/57: $45 (lo) & $00 or $FF (hi)
FFB9-   A9 BF       LDA  #$BF
FFBB-   48          PHA
FFBC-   A9 50       LDA  #$50          ; Setzt $BF50 (LANGIRQ) als RTI-Adresse
FFBE-   48          PHA
FFBF-   A9 04       LDA  #$04          ; und I-Flag im P-Reg (IRQ maskiert)
FFC1-   48          PHA
FFC2-   A9 FA       LDA  #$FA          ; Monitor BRK ($FA42)
FFC4-   48          PHA
FFC5-   A9 41       LDA  #$41            im Monitor-ROM steht auf $FFCB ein RTS,
FFC7-   48          PHA                  d.h. nach Umschalten wird „returnt"
FFC8-   8D 82 C0    STA  $C082         ; ROMON => „RTS" im Monitor
```

Zuerst wird (wie im Monitor auch) getestet, ob dieser IRQ durch eine BRK-Instruktion ausgelöst wurde. Ist das der Fall, wird $FA42 als „Returnadresse" auf den Stack gebracht. Durch das nach dem Schalten von ROMON als nächster Befehl folgende RTS (vgl. RESET) wird der IRQ-Handler des Monitors angesprungen. Hier wird noch einmal geprüft, ob der IRQ durch ein BRK ausgelöst wurde und dann entsprechend verzweigt.
Liegt dagegen ein „echter" IRQ vor, wird geprüft, welche Bank der Language Card im Moment des IRQ angeschaltet war und INTBANK1 ($BF8D) sowie INTBANK2 ($BF57) auf der Global Page entsprechend gesetzt. Damit erklärt sich, warum in INTEXIT ($BFD0) immer der Wert $01 als INTBANK gesetzt wird – innerhalb des Monitors existiert eine solche Routine nicht und innerhalb der Language Card wird der Wert von INTBANK1 immer neu gesetzt. Beim Boot enthält die Speicherstelle INTBANK1 ebenfalls den Wert $01.
Vor dem Setzen des Monitor-IRQ-Handlers ($FA42) als „Returnadresse" werden noch folgende Werte auf den Stack gebracht: $04 und $BF50.

Danach folgt auch für einen „echten" IRQ der Sprung zum IRQ-Handler des Monitors und von da aus derselbe Ablauf wie für ein „Monitor-IRQ".

Erst nach dem „RTI" von INTEXIT ($BFEA) geht es anders weiter:
„RTI" unterscheidet sich von „RTS" in zwei Punkten. Zum einen wird auch das P-Register vom Stack geholt, zum anderen wird hier die Returnadresse **nicht** noch einmal um eins erhöht.
Nach einem IRQ „in der Language Card" und dem „RTI" von INTEXIT hat das P-Register den Wert $04, d.h. IRQ's sind weiterhin gesperrt und es geht mit $BF50 weiter!

– LANGIRQ

```
BF50-   8D C0 8B    STA  $C08B          ; BANKSELECT: Bank 1 (kein WRITE)
BF53-   4C D8 FF    JMP  $FFD8          ; in der Language Card!
```

– LANGINTEXIT

```
FFD8-   8D 88 BF    STA   $BF88         ; INTAREG - unnötig, gerade geladen
FFDB-   AD 56 BF    LDA   $BF56         ; INTACCU
FFDE-   85 45       STA   $45           ; ACC des Monitors - RWTB-Pointer
FFE0-   AD 8B C0    LDA   $C08B         ; BANKSELECT von Bank 1
FFE3-   AD 8B C0    LDA   $C08B           + WRITE ENABLE
FFE6-   AD 57 BF    LDA   $BF57         ; nur $00 (Bank 1) oder $FF(Bank 2)
FFE9-   4C D3 BF    JMP   $BFD3         ; INTEXIT2 - schaltet die Bank zurück + RTI
```

In der IRQ-Behandlung sind zwei Speicherstellen äußerst undurchsichtig – INTBANK1 und 2 sowie die Speicherstelle $0045 (ACC). Fassen wir die Aktionen mit diesen Speicherstellen zusammen:

IRQ in der Language Card:

```
FF9B-   48          PHA                 ; Akku-Inhalt während des IRQ
FF9C-   A5 45       LDA  $45            ; ACC - RWTB-Pointer
FF9E-   8D 56 BF    STA  $BF56          ; INTACCU
FFA1-   68          PLA
FFA2-   85 45       STA  $45
        .....
FFB3-   8D 8D BF    STA  $BF8D          ; INTBANK1
FFB6-   8D 57 BF    STA  $BF57          ; INTBANK2
```

Der Akku-Inhalt während des IRQ steht in ACC, der vorherige Inhalt von ACC wurde auf $BF56 kopiert. Der Sprung zum IRQ-Handler des Monitors erfolgt – egal ob BRK oder „echter" IRQ – immer durch das Setzen der Returnadresse $FA42, d.h. hinter die Instruktion „STA $45" des „Monitor-IRQ".
INTBANK1 und INTBANK2 sind beide auf den gleichen Wert gesetzt – $00 (Bank 1) oder $FF (Bank 2). Bei einem IRQ via BRK sind diese Speicherstellen undefiniert, was niemanden stört – eine Rückkehr in ein laufendes Programm ist dann auch nicht vorgesehen.

Für einen „Monitor-IRQ" wird nach $FA40 gesprungen:

```
FA40-   85 45       STA  $45            ; Akku-Inhalt für „Monitor-IRQ"
FA42-   ......
```

Bei einem „echten" IRQ geht es via JMP ($03FE) mit $BFEB weiter, hier wird die Bank 1 eingeschaltet und zu $D13A gesprungen:

```
D13A-   A5 45       LDA  $45
D13C-   8D 88 BF    STA  $BF88          ; INTAREG - Akku-Inhalt während IRQ
        ......
```

Am Ende der Routine auf $D13A steht ein Sprung zu $BFD0, wo die entsprechende Bank bzw. das Monitor-ROM wieder eingeschaltet wird:

```
BFD0-   AD 8B BF    LDA  $BF8D          ; INTBANK1
BFD3-   F0 0D       BEQ  $BFE2          ; $00 = Bank 1 war/ist eingeschaltet
        ....
BFE2-   A9 01       LDA  #$01
BFE4-   8D 8D BF    STA  $BF8D          ; INTBANK1 auf $01
BFE7-   AD 88 BF    LDA  $BF88          ; INTAREG
BFEA-   40          RTI
```

INTBANK1 wird nur nicht auf $01 gesetzt, wenn ein „Monitor-IRQ" stattgefunden hat. In diesem Fall hat diese Speicherstelle sowieso den Wert $01.
Bei einem „Monitor-IRQ" springt der RTI zum laufenden Programm zurück – der Akku enhält dann auch wieder den korrekten Wert.
Bei einem „Language Card-IRQ" geht es via $BF50 mit $FFD8 weiter:

```
FFD8-   8D 88 BF    STA  $BF88          ; Unsinn, unnötig & verwirrend
FFDB-   AD 56 BF    LDA  $BF56          ; INTACCU
FFDE-   85 45       STA  $45            ; zurück in ACC bzw. RWTB-Pointer
        ......
FFE6-   AD 57 BF    LDA  $BF57          ; INTBANK2
```

Es folgt ein Sprung zurück zu INTEXIT2 ($BFD3). Hier muß INTBANK2 benutzt werden, weil INTBANK1 bereits vom „ersten Durchlauf" von $BFD0 auf den Wert $01 gesetzt wurde. Dieses Mal führt der RTI am Ende der Routine dann tatsächlich zum Rücksprung ins laufende Programm.

Dieser entsetzlich verwirrende Ablauf hat seinen Grund:
Der IRQ-Handler des Monitors ist nicht zu ändern, und hier wird die Speicherstelle $0045 benutzt und durch das Zwischenspeichern des Akku-Inhalts verändert.
Im MLI (Language Card eingeschaltet) kann diese Speicherstelle allerdings den höherwertigen Teil der Quell/Zieladresse für die RWTB enthalten – ein IRQ kurz vor oder während einer RWTB-Benutzung hätte damit katastrophale Folgen.

Trotzdem – angesichts dieses „Verhaus": Wie wäre es, die Unterscheidung von BRK und „echtem" IRQ innerhalb der Language Card etwas geradliniger zu nutzen?
Bei einem BRK würde ein Anschalten des Monitor-ROMs mit anschließender Benutzung der Monitor-Routinen folgen, bei einem IRQ dieselbe Routine ($D13A) wie bereits implementiert – ohne die Benutzung von ACC. Der „Monitor-IRQ" bekäme einen etwas früheren Einsprung in dieselbe Routine, bei dem von ACC in eine unkritische Speicherstelle kopiert würde. Das wär's bereits.

Dieser Vorschlag übersieht allerdings großzügig ein Grundübel, das die Implementation anderer RWTB's in der von Apple, Inc. vorgeschlagenen Form unmöglich macht:

Da die Unterscheidung zwischen Monitor und MLI nicht etwa auf dem Flag „MLI aktiv" basiert, sondern auf der Tatsache, daß die Language Card eingeschaltet ist, gibt das mit sämtlichen RAM Disks (IBS, Saturn etc.) ein Betriebssystem, bei dem man die vielgepriesenen Interrupts lieber ein für allemal abschalten sollte.
Begründung: Die erwähnten RAM-Karten belegen denselben Speicherbereich wie die Language Card, die Language Card muß also abgeschaltet werden. Folge: Die im RWTB-Übergabeprotokoll definierte Speicherstelle $45 wird bei einem IRQ verändert.

Auch für die RWTBs anderer Speichermedien gibt es Ärger, selbst wenn man hier die Routinen in die Bank 2 der Language Card „hineinpackt": Der INTEXIT schaltet zwar diese Bank wieder ein, „vergißt" aber, auch WRITE ENABLE zu setzen. Folge: Selbstmodifizierender Kode ist unmöglich und die Variablen einer solchen RWTB können auch nicht in dieser Bank stehen.

Es bleibt natürlich abzuwarten, ob sich dieser Mißstand in den nächsten Versionen von ProDOS ändern wird – bei der Version 1.0.2 ist jedenfalls noch nichts in dieser Richtung passiert.

## 3.2. SET TIME für die Thunderclock

Die MLI-Funktion $82 – SET TIME bewirkt im MLI selber ein „JSR $BF06". Auf $BF06 ist per „default" ein RTS gesetzt.
Wird beim Boot von PRODOS eine Thunderclock-Uhrenkarte (oder eine Kompatible) identifiziert, wird aus dem „RTS" auf $BF06 ein „JMP $F142" (für 48k: JMP $B142).

Die folgende Routine liest die momentane System-Zeit und setzt die Speicherstellen $BF90..$BF93 entsprechend. Der RTS am Ende der Routine führt (bei regulärem Aufruf) wieder ins MLI – entweder zu $D078 (interner MLI-Exit), wenn nur SET TIME aufgerufen wurde oder zum Command Handler für die $C0-$D3 Commands, von dem aus bei verschiedenen Funktionen (CREATE, CLOSE etc.) die System Time benötigt wird.

Die Routine wird vom MOVER innerhalb des Boot immer auf $F142 (64k-Maschine) kopiert, egal, ob die Routine, die die Slots absucht, eine Uhrenkarte identifiziert hat oder nicht.

Bei einer 48k-Maschine wird die Routine nicht nur kopiert, sondern auch speicherplatzmäßig angepaßt („verschoben"), sie beginnt dann auf $B142.
Darüberhinaus setzt die Routine, die die Slots absucht, die Slotnummer der Uhrenkarte (falls gefunden) im Parameterblock des MOVER. Für beide Speichergrößen (48k und 64k) wird die Clock-Routine mit dem MOVER ein zweites Mal relokalisiert, dieses Mal ohne Verschiebung der Routine selber – das Ergebnis ist eine Anpassung der beiden JSRs zur Uhrenkarte auf die Slotadresse der gefundenen Karte. Die dabei veränderten Bytes sind im Listing durch Sternchen ersetzt.

```
F142-   AE 50 F1    LDX   $F150         ; $Cx, ist relociert worden
F145-   BD 38 05    LDA   $0538,X       ; I/O-Zwischenspeicher-Byte für die
F148-   48          PHA                   Card-Save
F149-   A9 A3       LDA   #$A3          ; Befehl: Time im ProDOS-Format
F14B-   20 0B **    JSR   $C10B         ; OUT des Befehls
F14E-   20 08 **    JSR   $C108         ; IN liefert „MM-YY-DD hh:mm" in ASCII
F151-   18          CLC                   (Monat-Jahr-Tag-Std-Min) auf $200..$20D
F152-   A2 04       LDX   #$04
F154-   A0 0C       LDY   #$0C          ; Start von hinten: Minuten zuerst
F156-   B9 00 02    LDA   $0200,Y       ; Zehnerstelle
F159-   29 07       AND   #$07          ; Begrenzung auf 0..7
F15B-   85 3A       STA   $3A
F15D-   0A          ASL
F15E-   0A          ASL                 ; Zehner-Wert * 4
F15F-   65 3A       ADC   $3A           ; + einmal den Wert = Zehner-Wert * 5
F161-   0A          ASL                 ; * 2 = Zehner-Wert * 10
F162-   79 01 02    ADC   $0201,Y       ; + niederwertige Stelle
F165-   38          SEC
F166-   E9 B0       SBC   #$B0          ; -$B0 („0" = $B0 in ASCII)
F168-   95 3A       STA   $3A,X         ; $3E:mm/$3D:hh/$3C:$DD/$3B:YY/$3A:MM
F16A-   88          DEY
F16B-   88          DEY                 ; Überspringen von -, ,: zwischen den
F16C-   88          DEY                   einzelnen Werten
F16D-   CA          DEX
F16E-   10 E6       BPL   $F156

F170-   A8          TAY                 ; Monat (Bereich: $01 bis $0C):
F171-   4A          LSR
F172-   6A          ROR
F173-   6A          ROR                 ; MMMM wird zu MMM0 0000, das MSB bleibt
F174-   6A          ROR                   im Carry
F175-   05 3C       ORA   $3C           ; ORA „Day"
F177-   8D 90 BF    STA   $BF90         ; $BF90: MMM DDDDDD (Monat-Tage)
F17A-   08          PHP                 ; MSB von „Monat"
F17B-   29 1F       AND   #$1F          ; ergibt 000 DDDDD („Day")
F17D-   79 AB F1    ADC   $F1AB,Y       ; Y= „Monat", ADC mit Carry = MSB „Monat"!
F180-   90 02       BCC   $F184
F182-   69 03       ADC   #$03          ; +4: nur für „Monat" = 9 und „Day" > 8
F184-   38          SEC
F185-   E9 07       SBC   #$07
F187-   B0 FC       BCS   $F185
F189-   69 07       ADC   #$07          ; Ergebnis zwischen 1 und 7
F18B-   E5 3B       SBC   $3B           ; minus YEAR
F18D-   B0 02       BCS   $F191
F18F-   69 07       ADC   #$07
F191-   A8          TAY                 ; wieder zwischen 1 und 7
F192-   B9 B8 F1    LDA   $F1B8,Y       ; Tabelle mit Jahreszahlen
F195-   28          PLP                 ; MSB von „Monat"
F196-   2A          ROL                   geht auf Bit 0 im Jahresbyte
F197-   8D 91 BF    STA   $BF91         ; Jahre („YY") + MSB vom „Monat" in Bit 0
F19A-   A5 3D       LDA   $3D           ; Hours – Stunden („hh")
F19C-   8D 93 BF    STA   $BF93
```

```
F19F-   A5 3E       LDA   $3E           ; Minuten („mm")
F1A1-   8D 92 BF    STA   $BF92

F1A4-   68          PLA                 ; Restore der I/O-Zwischenspeicherstelle
F1A5-   AE 50 F1    LDX   $F150         ; $Cx
F1A8-   9D 38 05    STA   $0538,X
F1AB-   60          RTS

F1AC- 00 1F 3B 5A 78 97 B5  D3 F2 14 33 51      ; Korrekturfaktoren
F1B8- 54 54 53 52 57 56 55      ; Jahreszahlen: 84/84 83 82 87 86 85
```

Was die Umrechnung der Jahreszahlen angeht:
Hier haben wir ein schönes Beispiel dafür, wie man ein Listing kommentieren kann, das man selber nicht versteht. Diese „Umrechnung" funktioniert offensichtlich nur für die Jahre '82 bis '87 – was die Thunderclock dabei tatsächlich an Jahreszahlen liefert bzw. wieso diese Rechnerei nötig ist, entzieht sich meiner Kenntnis.

# Teil IV – Die System Global Page

## 4.1. Listing und Beschreibung der SGP für 64k und „128k"-Maschinen

Die System Global Page (SGP) stellt das Bindeglied zwischen dem MLI und dem „user"-Programm dar. Sämtliche Funktionen von ProDOS sowie alle Werte, die ein „user"-Programm ohne den MLI-Aufruf einer MLI-Funktion abrufen kann, sind hier verzeichnet.
Apple, Inc. hat im „Technical Manual" allerdings eine weitere Unterscheidung vorgenommen: Es gibt Speicherstellen der SGP, die vom „user"-Programm aus gelesen, teilweise sogar beschrieben werden dürfen, und solche, die nicht fest definiert sind und „for informational purpose only" gelistet sind.

Innerhalb des Files PRODOS sind zwei vorbereitete System Global Pages gespeichert (64k: $4E00-$4EFF, 48k: $4F00-$4FFF). Je nachdem, wie die Prüfung auf „64k" ausfällt, wird einer der beiden Speicherbereiche auf $BF00-$BFFF kopiert.
Auf die Unterschiede zwischen „48k" und „64/128k" wird im nächsten Kapitel eingegangen.

```
BF00-  4C B7 BF    MLIENTRY: JMP  $BFB7      ; „offiziell"
```

Das ist der einzige definierte Einsprung für einen MLI-Aufruf. Das Ziel dieses Sprungs liegt zwar wieder innerhalb der SGP, $BFB7 gehört aber zu den „inoffiziellen" Teilen und sollte daher nicht benutzt werden.

```
BF03-  4C CF EE    REBOOT:    JMP  $EECF      ; „inoffiziell"
```

Dieser Sprung führt zum Command Handler für $65 – REBOOT. Wird vom MLI Command Handler her mit JSR(!) angesprungen, wenn das Command $65 mit vier „Dummy"-Parametern gegeben wurde. Wenn dieser Vektor direkt vom „user" benutzt werden soll, muß vorher die Bank 1 der Language Card angeschaltet werden.
Dieser Vektor hat eine etwas merkwürdige Vorgeschichte: innerhalb des Files PRODOS steht hier ein Sprung zu $BFF6 (SYSTEM DEATH2). Diese Zieladresse wird vor dem Installieren von REBOOT in der Bank 2 auf $EECF gesetzt.

```
BF06-  4C 42 F1    SETTIME:   JMP  $F142      ; „offiziell" + WRITE
```

Dieser Sprung führt zum Handler für das Command $82 – SET TIME. Im File PRODOS steht auf $4E06 (=> $BF06) ein RTS. Wird während der Slot-Absuche eine Thunderclock-

Uhrenkarte (oder eine „Kompatible") gefunden, wird anstelle des RTS ein „JMP" gesetzt. Hier kann man für andere Uhrenkarten seine eigene Startadresse hineinpoken (wenn man die eigene Uhrenroutine nicht gleich auf $F142 verlegt). Auch hier muß bei einem direkten Aufruf natürlich vorher die Bank 1 der Language Card eingeschaltet werden.

```
BF09-  4C DA D1     SET-SYSERR: JMP  $D1DA     ; „inoffiziell"
```

Wird von verschiedenen Stellen des MLI aufgerufen und setzt die Speicherstelle SYSERR mit der jeweiligen Fehlernummer. Die Routine $D1DA holt danach eine Returnadresse vom Stack und führt ein RTS aus, d.h. wenn $BF09 mit JSR aufgerufen wird, ist damit gleichzeitig die Routine, die $BF09 aufgerufen hat, beendet. Innerhalb des MLI führt dieses zweite RTS immer zu $D078, dem internen MLI-Exit (Ausnahme: $D0A2 – NO DEVICE (RTS zur Routine, die DO DISK I/O aufgerufen hat) und $D0A2 (hier folgt ein Sprung zu $D078).

```
BF0C-  4C E6 D1     SYSDEATH:  JMP  $D1E6      ; „inoffiziell"
```

Ein Sprung hierher führt zum SYSTEM DEATH – der Ausgabe von „INSERT SYSTEM DISK AND RESTART -ERRxx". Wird von innerhalb des MLI angesprungen, wenn etwas „Unerklärliches" passiert, z.B. wenn ein offener File plötzlich keinen Filepuffer mehr hat. Der einzige plausible Fehler, der zum SYSTEM DEATH führt, ist ein Interrupt, für den sich keine der Service-Routinen zuständig fühlt, alle anderen Sprünge zu SYSTEM DEATH sind logisch gesehen unmöglich.
Die Fehlernummern sind im „Technical Manual" nicht dokumentiert und hier in Anhang F aufgelistet.

```
BF0F-  **           SYSERR:    DS 01           ; „offiziell"
```

SYSERR wird beim Einsprung in das MLI auf $00 gesetzt ($D01A) und beim internen Exit ($D07D) zurückgeladen. Ist diese Speicherstelle zwischendurch gesetzt worden (nur via SET SYSERR), wird das Carry vor dem Verlassen des MLI gesetzt (ERROR), ansonsten wird das MLI mit Carry gelöscht und der „Fehler"nummer $00 im Akku verlassen.
Bei den internen Aufrufen zum Lesen/Schreiben eines Blocks (Directory/VBM-Blocks: $DDF0 oder Blocks eines Files: $E073) wird SYSERR ebenfalls auf $00 gesetzt. Wieso, ist eigentlich nicht einzusehen: die RWTB setzt SYSERR im Fehlerfall nicht, sondern kehrt mit gesetztem Carry und der ERR# im Akku zurück.

```
BF10-  A2 D0        DRV1TBL:   DW  $D0A2       ; Slot 0, Drive 1 – „offiziell"
BF12-  A2 D0                   DW  $D0A2       ; Slot 1, Drive 1   und WRITE
BF14-  A2 D0                   DW  $D0A2       ; Slot 2, Drive 1
BF16-  A2 D0                   DW  $D0A2       ; Slot 3, Drive 1
BF18-  A2 D0                   DW  $D0A2       ; Slot 4, Drive 1
BF1A-  A2 D0                   DW  $D0A2       ; Slot 5, Drive 1
BF1C-  00 F8                   DW  $****       ; Slot 6, Drive 1, hier DISKII
BF1E-  A2 D0                   DW  $D0A2       ; Slot 7, Drive 1

BF20-  A2 D0        DRV2TBL:   DW  $D0A2       ; Slot 0, Drive 2 – „offiziell"
BF22-  A2 D0                   DW  $D0A2       ; Slot 1, Drive 2    und WRITE
BF24-  A2 D0                   DW  $D0A2       ; Slot 2, Drive 2
BF26-  00 FF                   DW  $****       ; Slot 3, Drive 2, hier RAMDisk
BF28-  A2 D0                   DW  $D0A2       ; Slot 4, Drive 2

BF2A-  A2 D0                   DW  $D0A2       ; Slot 5, Drive 2
BF2C-  00 F8                   DW  $****       ; Slot 6, Drive 2, hier DISKII
BF2E-  A2 D0                   DW  $D0A2       ; Slot 7, Drive 2
```

Diese beiden DRIVE TABLEs enthalten die Startadressen der jeweiligen RWTBs. Im Gegensatz zur DEVICE TABLE sind diese Startadressen nach Slotnummern geordnet. Wenn die Slot-Suchroutine innerhalb des Relocators einen Disk-Controller identifiziert, wird hier die Startadresse der RWTB eingetragen. Sie sollte sich aus Slotadresse ($Cx) und dem Byte $CxFF zusammensetzen. Allerdings sind die beiden „üblichen" Drives (DISKII und RAMDisk) hier die Ausnahme: Wird eine DISKII identifiziert, landet die Startadresse $F800 (48k: $B800) an der entsprechenden Stelle, für „128k" (RAMDisk) wird die Startadresse $FF00 eingetragen. Die RAMDisk ist dabei immer als „Slot 3, Drive 2" definiert.

Adressiert wird die DRIVE TABLE innerhalb des MLI nur von DO DISK I/O ($D0DA). Die Unitnummer (DSSS 0000) wird durch 8 dividiert (=> 000D SSS0, entspricht „D" * $10 + „SSS" * 2 ) und ergibt den Index zur Basis $BF10.

Die innerhalb des Files PRODOS vordefinierte SGP enthält nur die Adresse $D0A2. Die entsprechende Routine setzt ERR#28: NO DEVICE CONNECTED und kehrt via SET SYSERR zurück zu der Routine, die DO DISK I/O aufgerufen hat.

```
BF30-  **          LASTDEV:   DS  1          ; „offiziell"
```

Hier wird per definitionem die Unitnummer des zuletzt benutzten DEVICE (= Unit) gespeichert. Auch wenn diese Speicherstelle innerhalb des MLI des öfteren auch zum Setzen der Unitnummer vor der Benutzung eines Units (und damit nicht als „last recently used", sondern als „to be used soon") „mißbraucht" wird, enthält sie nach dem Verlassen des MLI mit Sicherheit die Nummer des zuletzt adressierten Units.

```
BF31-  FF          DEVNUM:    DFB  $FF       ; „offiziell" + WRITE
```

Enthält die Anzahl der DEVICES minus eins, also den Index zum letzten DEVICE innerhalb der DEVICE TABLE. In der vordefinierten SGP im File PRODOS ist diese Nummer auf $FF (= 00 DEVICES) gesetzt und wird für jedes gefundene DEVICE (Slotabsuche und Installation der RAMDisk bei „128k") entsprechend erhöht.

Wenn weitere DEVICES vom „user" installiert werden, muß diese Nummer entsprechend erhöht werden.

Innerhalb des MLI wird auf diese Speicherstelle nur für die beiden ONLINES zugegriffen. Das Command ONLINE („ALL") ruft die Routine $DA57 auf, wo die DEVICE TABLE kopiert wird; Das interne ONLINE (Suche nach einem Volume) benutzt diese Routine ebenfalls.

```
BF32-  00          DEVTBL:    DFB  $00       ; „offiziell" + WRITE

BF33-  00                     DFB  $00
BF34-  00                     DFB  $00
BF35-  00                     DFB  $00
BF36-  00 00 00 00 00         DFB  $00,00,00,00,00
BF3B-  00 00 00 00 00         DFB  $00,00,00,00,00
```

Die DEVICE TABLE enthält Platz für insgesamt 14 DEVICES, d.h. Speichermedien. Das ist mehr als reichlich: wenn mehr als 8 DEVICES besetzt sind, führt das auch bei einem ONLINE zu ERR#55: VCB FULL – im VCB ist Platz für maximal 8 Volumes.

In der vordefinierten SGP im File PRODOS ist diese Tabelle leer, während der Slotabsuche werden gefundene DEVICES hier verzeichnet. Jedes Byte in der DEVICE TABLE steht für ein DEVICE (d.h. zwei Drives im selben Slot belegen zwei Byte in der DEVICE TABLE). Das Byte hat die Form DSSS TTTT:

- „D" = Drive, (0 oder 1 für Drive 1 bzw. 2)
- „SSS" = Slot (0 bis 7, Slot 0 ist allerdings nicht möglich)
- „TTTT" = Typ des DEVICES, wird aus dem DRIVE CHARACTERISTICS (Byte $CxFE auf der Controller-Karte) zusammengesetzt. Bis jetzt sind drei Typen definiert:

0000 = DISKII / 0100 = ProFile-Harddisk / 1111 = RAMDisk

Aus „DSSS" ergibt sich der entsprechende Index in der DRIVE TABLE, der DEVICE-Typ wird innerhalb des MLI nicht ausgewertet.

Direkte Zugriffe auf die DEVICE TABLE erfolgen nur für ONLINE (intern und explizit). In den entsprechenden Routinen wird dann aus „DSSS" und Maskieren von „TTTT" die Unitnummer.

```
BF40-  43 4F ....  COPR: ASC 'COPR. APPLE.1983'
```

Zugriffe auf diese Speicherstellen finden innerhalb des MLI nicht statt.

```
BF50-  8D 8B C0    LANGIRQ:   STA  $C08B      ; „inoffiziell"
BF53-  4C D8 FF               JMP  $FFD8
```

Passiert ein IRQ, während die Language Card angeschaltet ist („MLI aktiv"), wird im IRQ-Handler ($FF9B) als „RTI-Adresse" $BF50 gesetzt. Auf $FFD8 wird dann die (durch den IRQ-Handler des Monitors veränderte) Speicherstelle $0045 (RWTB-Pointer!) wieder in Ordnung gebracht (s. Kapitel „Interrupts").

```
BF56-  **          INTACCU:   DS  1           ; „inoffiziell"
```

Zwischenspeicher für den Inhalt der Speicherstelle $0045 bei einem IRQ, während die Language Card angeschaltet ist.

```
BF57-  **          INTBANK2:  DS  1           ; „inoffiziell"
```

Zweite Speicherstelle für einen „Marker" der Bank, die zur Zeit eines IRQ angeschaltet war. Wird bei einem IRQ nur gesetzt, wenn zum Zeitpunkt des IRQ die Language Card angeschaltet ist. Grund: Der INTEXIT wird dann zweimal durchlaufen und setzt beim ersten Mal bereits die „normale" Speicherstelle (INTBANK1) auf „Monitor" (s. „Interrupts").

```
BF58-  C0          SBITMAP:   DFB  $C0          ; „offiziell" + WRITE
BF59-  00 00 00 00            DFB  $00,00,00,00
BF5D-  00 00 00 00            DFB  $00,00,00,00
BF60-  00 00 00 00            DFB  $00,00,00,00
BF64-  00 00 00 00            DFB  $00,00,00,00
BF68-  00 00 00 00            DFB  $00,00,00,00
BF6C-  00 00 00               DFB  $00,00,00

BF6F-  01                     DFB  $01
```

Die SYSTEM BIT MAP ist das Verzeichnis der freien/belegten Speicherseiten innerhalb der unteren 48k RAM. Jede Speicherseite ($100 Byte) wird durch ein Bit repräsentiert, ein gesetztes Bit steht für „Speicherseite belegt".
Die Reihenfolge der Bits innerhalb eines Bytes ist „verkehrt" herum, d.h. Bit 0 steht für die höchste, Bit 7 steht für die niedrigste von 8 Speicherseiten.
Die Reihenfolge der Bytes innerhalb der Tabelle entspricht der Reihenfolge der Speicheradressen, d.h. das erste Byte ($BF58) steht für die Speicherseiten $0000 bis $07FF, das zweite für die Speicherseiten $0800 bis $0FFF etc.
Das letzte Byte der SBM steht somit für die Speicherseiten $B800 bis $BFFF.

In der vordefinierten SGP im File PRODOS sind (wie hier aufgelistet) nur die Speicherseiten 0 (Zeropage) und 1 (Stack) sowie die Seite $BF (System Global Page selber) belegt.

REBOOT gibt den gesamten Speicherbereich bis auf die Seite $BF und die Bereiche $0000-$01FF sowie $0400-$07FF wieder frei. Jedes Programm sollte vor der Benutzung eines Speicherbereichs zuerst gegen die SBM prüfen.

Innerhalb des MLI wird für alle Funktionen, die Bereiche außerhalb des MLI selber beschreiben bzw. belegen (Filepuffer), gegen die SBM geprüft. Filepuffer werden beim OPEN eines Files in der SBM belegt und bei einem CLOSE wieder freigegeben. Für BLOCKREAD/WRITE wird ebenfalls gegen die SBM geprüft (s. „Spezielles").
Für das Zurückliefern von Werten (Schreiben in einen „user"-Parameterblock) wird nicht gegen die SBM geprüft (Ausnahme: GET PREFIX und ONLINE, hier werden die entsprechenden Zielpuffer getestet).

```
BF70-  00 00        BUFTBL:    DW  $0000       ; „offiziell"
BF72-  00 00                   DW  $0000
BF74-  00 00 00 00             DW  $0000, 0000
BF78-  00 00 00 00             DW  $0000, 0000
BF7C-  00 00 00 00             DW  $0000, 0000
```

Die BUFFER TABLE enthält die Startadressen der Filepuffer offener Files. Der FCB-Eintrag eines Files enthält einen Index, der immer identisch mit der Reference Number des Files mal 2 ist (Bereich: $02 bis $10). Deshalb erfolgt die Adressierung der BUFFER TABLE auch immer mit der Basis $BF6E bzw. $BF6F für das höherwertige Byte der Startadresse.

Gesetzt wird ein Pufferstart von OPEN (und SET BUF), gelesen wird er von diversen Routinen (GET BUF, Set File Data, Restore der Filepuffer-Pointer nach READ, Stufen 2 und 3 etc.), gelöscht wird in der BUFFER TABLE nur von CLOSE oder bei einem Fehler innerhalb von OPEN.
Alle Lese-Zugriffe auf die BUFFER TABLE erfolgen durch die Routine $EE26 (Kopieren einer Puffer-Startadresse in den Variablenbereich).
Gelöscht wird durch DEALLOCATE FILEBUFFER ($EE34), gesetzt durch die Routine auf $EDD7.

```
BF80-  00 00        INTRUPTS:  DW  $0000       ; Service 1 - „offiziell"
BF82-  00 00                   DW  $0000       ; Service 2
BF84-  00 00                   DW  $0000       ; Service 3
BF86-  00 00                   DW  $0000       ; Service 4
```

Diese Tabelle enthält die Startadressen der einzelnen Interrupt-Services. Ein Eintrag mit Hibyte = $00 gilt als nicht besetzt.
Verändert wird diese Tabelle nur durch ALLOCATE und DEALLOCATE INT. Da hier auch mit Reference-Nummern im Bereich 1-4 gearbeitet wird, erfolgt die Adressierung der INTERRUPT TABLE hier mit der Basis $BF7E bzw. $BF7F für das Hibyte der Startadressen.
Im Falle eines IRQ werden die einzelnen Startadressen beginnend mit $BF80/81 auf „besetzt" geprüft und ggf. der entsprechende Service aufgerufen.
Hier findet sich auch der einzige plausible SYSTEM DEATH: Tritt ein IRQ auf, bevor der entsprechende Service in der INTERRUPT TABLE eingetragen ist, „verabschiedet" sich das System nach dem erfolglosen Absuchen der gesamten Tabelle mit SYSTEM DEATH und ERR#01 (für die Version 1.0.2 wurde diese Fehlernummer auf #02 geändert, s. „neue Versionen").

```
BF88- **        INTAREG:   DS  01        ; „offiziell"
BF89- **        INTXREG:   DS  01
BF8A- **        INTYREG:   DS  01
BF8B- **        INTSREG:   DS  01
BF8C- **        INTPREG:   DS  01
```

Zwischenspeicherplatz für A, X, Y, Stackpointer und Flag-Register bei einem Interrupt. Werden nur in der IRQ-Verarbeitung benutzt. Nach erfolgreichem Service werden die Register wieder zurückgeladen.

```
BF8D- 01        INTBANK1:  DFB  01        ; „inoffziell"?
```

Diese Speicherstelle enthält einen „Marker" für die z.Zt. des IRQ aktive Bank der Language Card (Bank 1 = $00 / Bank 2 = $FF) oder für den Monitor ($01).
Da die IRQ-Verarbeitung des Monitors nicht auf ProDOS eingerichtet ist und der entsprechende Handler deshalb diese Speicherstelle auch nicht setzt, ist hier in der vorbereiteten SGP im File PRODOS der Wert $01 definiert (=> INTBANK = Monitor-ROM).
Die Routine INTEXIT setzt INTBANK1 nach dem Zurückschalten der Bank immer auf $01: im Falle eines IRQ „innerhalb der Language Card" wird diese Speicherstelle entsprechend gesetzt ($FF9B), falls der IRQ „im Monitor" passiert, bleibt es dabei.
Die Bezeichnung INTBANKID im „Technical Manual" ist irreführend (die BANKIDs haben die Werte $D8 (Bank1) bzw. $EE (Bank2)), die Angabe des Wertes $00 trifft für die ProDOS-Versionen 1.0.1 und 1.0.2 nicht zu.

```
BF8D- ** **     INTADDR:   DS  02        ; „offiziell"
```

Hier wird die Returnadresse zum laufenden Programm im Falle eines IRQ gespeichert.

```
BF90- ** **     DATE:      DS  02        ; „offiziell" + WRITE
BF92- ** **     TIME:      DS  02
```

Diese Speicherstellen enthalten die System Time in der folgenden Form:

```
$BF90/91: FEDCBA98 76543210     $BF92/93: FEDCBA98 76543210
          YYYYYYYM MMMDDDDD               hhhhhhhh mmmmmmmm
```

„Y" = Jahr, Bereich 00 bis 127 (entspricht 1900 bis 2027)
„M" = Monat; „D" = Tag; „h" = Stunde; „m" = Minute.

Gesetzt werden diese Speicherstellen von SET TIME, gelesen werden sie innerhalb des MLI von jeder Funktion, die einen Directory-Eintrag macht oder ändert (Ausnahme: RENAME sowie SET FILE INFO mit spezifiziertem DATE & TIME im Parameterblock).

```
BF94- **          LEVEL:    DS  01          ; „offiziell"
```

Der Wert dieser Speicherstelle wird bei einem OPEN in den entsprechenden FCB-Eintrag kopiert. Bei einem CLOSE „ALL" werden Files übergangen, deren internes LEVEL kleiner als das LEVEL auf der SGP zum Zeitpunkt des CLOSE-Befehls ist. Wann diese Speicherstelle verändert wird, ist nur durch eine Analyse des BASIC.SYSTEM herauszubekommen (s. „Spezielles").

```
BF95- **          BUBIT:    DS  01          ; „offiziell"
```

Bei dieser Speicherstelle spielt nur der Wert von Bit 5 (= „$20") eine Rolle. Jeder interne MLI-Exit ($D078) setzt diese Speicherstelle auf $00. Gelesen wird sie nur in SET FILE INFO, wo durch eine Verknüpfung dann das MLI-interne BACKUP-Bit ($F074) gesetzt/gelöscht wird. Hier gilt dasselbe wie für LEVEL.

```
BF96- 00 00       SPARE:    DFB  $00,00     ; „offiziell"
```

Für zukünftige Erweiterungen reserviert.

```
BF98- **          MACHID:   DS  01          ; „offiziell"
```

Enthält ein Identifikationsbyte des momentan benutzten APPLE-Typs einschließlich eventueller Zusatzkarten und wird vom Relocator gesetzt. Die einzelnen Bits bedeuten:

| | | |
|---|---|---|
| Bit 7/6: | APPLE-Typ: | 00 = II / 01 = II+ / 10 = IIe / 11 = III-Emulation |
| Bit 5/4: | RAM: | 01 = 48k / 10 = 64k / 11 = 128k (RAMDisk) |
| Bit 3/2: | „reserviert", Bit 3 wird im Relocator behandelt (s.„Spezielles") | |
| Bit 1: | wenn gesetzt: 80-Col Karte (IIe/c) gefunden | |
| Bit 0: | wenn gesetzt: Thunderclock gefunden | |

Diese Speicherstelle wird innerhalb des MLI nicht gelesen, aber z.B. vom STARTUP der USERS.DISK abgefragt (Unterscheidung auf Groß- und/oder Kleinschreibung in den Erklärungen).

```
BF99- **          ROMBIT:   DS  01          ; „offiziell"
```

Innerhalb dieses Bytes steht jedes Bit für einen Slot, Bit 7 steht für Slot 7, Bit 6 steht für Slot 6 etc.
Wird bei der Slotabsuche innerhalb des Relocators ein besetzter Slot gefunden („bekannte" Karte oder konstanter READ von diesem Slot), wird das entsprechende Bit in ROMBIT gesetzt. Durch die Art des Tests werden auch RAM-Karten in ROMBIT eingetragen. Wird innerhalb des MLI nicht gelesen.

```
BF9A- **          PREFIX:   DS  01          ; „offiziell"
```

Enthält entweder den Startindex des momentanen System PREFIX in der Speicherseite $F100 oder den Wert $00 (kein PREFIX). Könnte bei der momentanen MLI-Version auch

via POKE auf $00 gesetzt werden (lösche PREFIX), der Befehl oPREFIX /" bewirkt aber dasselbe und ist darüberhinaus „zukunftssicher".

```
BF9B- **          MLIACTIVE: DS  01          ; „offiziell"
```

Beim „Betrieb" enthält diese Speicherstelle das Bit-Muster 0101 ...
Bei jedem MLI-Aufruf via $BF00 wird das Carry gesetzt und in Bit 7 dieser Speicherstelle hineinrotiert, beim internen Exit des MLI ($D078) wird mit „LSR" das Bit 7 auf $00 gesetzt. Da bei einem normalen Systemstart bereits zum Laden des „.SYSTEM"-Programmes mehrere MLI-Aufrufe erfolgt sind, hat diese Speicherstelle normalerweise den Wert $55 (0101 0101), wenn von außerhalb des MLI (PEEK oder Monitor) abgefragt wird.
Diese Speicherstelle wird innerhalb des MLI nicht benutzt – man könnte (sollte!) sie für die IRQ-Behandlung verwenden (s. „Interrupts").

```
BF9C- ** **       MLIRTS:    DS  02          ; „inoffiziell"
BF9E- **          XSAVE:     DS  01
BF9F- **          YSAVE:     DS  01
```

Hier werden die (erhöhte) Returnadresse zum „user"-Programm, X- und Y-Register beim Einsprung ins MLI ($D001/$D004 bzw. $D015) gespeichert und beim internen Exit ($D078) wieder zurückgeladen. Der Stackpointer bleibt unverändert, der Akku enthält die ERR# $00 (= kein Fehler) und das P-Register (Carry/Z-Flag) ist entsprechend gesetzt.

```
BFA0- 4D 00 E0    MLIEXIT:   EOR  $E000      ; „inoffiziell"
BFA3- F0 05                  BEQ  $BFAA      ; Bank 1 war/ist an: zweiter Test
BFA5- 8D 82 C0               STA  $C082      ; ROMON => Monitor bzw. APPLESOFT
BFA8- D0 0B                  BNE  $BFB5      ; „always" nach ROMON
BFAA- 8D F5 BF               LDA  $BFF5      ; BANKSEL 2
BFAD- 4D 00 D0               EOR  $D000      ; war wirklich Bank 1 vorher an?
BFB0- F0 03                  BEQ  $BFB5      ; Exit wenn ja

BFB2- AD 83 C0               LDA  $C083      ; Select von Bank 2

BFB5- 68                     PLA             ; ERR#
BFB6- 40                     RTI             ; zum „user"-Programm
```

Vor dem Sprung zu $BFA0 wird im internen Exit des MLI ($D078) die Returnadresse zum „user"-Programm von MLIRTS geladen und gepusht. Danach werden das P-Register und die ERR# ebenfalls auf den Stack gebracht. Der Akku wird mit der Speicherstelle BANKSEL (hier wurde beim Aufruf des MLI das ID ($D000) der aktiven Bank gespeichert) geladen. Nach dem Zurückschalten auf die Bank (1, 2 oder Monitor), die vor dem MLI-Aufruf an war, wird die ERR# wieder vom Stack geholt. Der „RTI" bewirkt sowohl ein „Pull P-Register" als auch das Return zum „user"-Programm.

Die BANKIDs:

| | |
|---|---|
| Monitor bzw. ROM: | $D000 = $6F, $E000 = $4C |
| Bank 1: | $D000 = $D8, $E000 = $20 |
| Bank 2: | $D000 = $EE, $E000 = $00 |

```
BFB7- 38          MLIENT1:   SEC             ; „inoffiziell"
BFB8- 6E 9B BF               ROR  $BF9B      ; setzt Bit 7 von MLIAKTIV
BFBB- AD 00 E0               LDA  $E000
BFBE- 8D F4 BF               STA  $BFF4      ; BANKSEL – $E000 der aktiven Bank
```

```
BFC1- AD 00 D0          LDA  $D000
BFC4- 8D F5 BF          STA  $BFF5      ; BANKSEL1 - $D000 d. aktiven Bank

BFC7- AD 8B C0          LDA  $C08B      ; SELECT der Bank 1
BFCA- AD 8B C0          LDA  $C08B      ; und WRITE ENABLE
BFCD- 4C 00 D0          JMP  $D000      ; Sprung zum MLI
```

$BFB7 wird von $BF00 her angesprungen (JMP). Hier werden die Speicherstellen $BFF4 und $BFF5 (BANKSEL/BANKSEL1) mit den Speicherstellen $E000 und $D000 der momentan aktiven Bank beladen, bevor die Bank 1 angeschaltet wird. Dieses Umkopieren ist notwendig, damit im MLIEXIT wieder auf die vorherige Speicherkonfiguration zurückgeschaltet werden kann.

```
BFD0- AD 8B BF    INTEXIT: LDA  $BF8D      ; INTBANK1 - „inoffiziell"
BFD3- F0 0D       INTLC:   BEQ  $BFE2      ; Bank 1 ist/war an
BFD5- 30 08                BMI  $BFDF      ; Bank 2 war an - Umschalten
BFD7- 4A                   LSR
BFD8- 90 0D                BCC  $BFE7      ; „never" - nur 48k (s. dort)

BFDA- AD 82 C0             LDA  $C082      ; ROMON
BFDD- B0 08                BCS  $BFE2      ; „always" nach ROMON: Exit
BFDF- AD 83 C0             LDA  $C083      ; BANKSELECT der Bank 2
BFE2- A9 01                LDA  #$01       ; und Setzen von „Monitor-IRQ"
BFE4- 8D 8D BF             STA  $BF8D        als INTBANK1
BFE7- AD 88 BF             LDA  $BF88      ; INTAREG - Akku Inhalt bei IRQ
BFEA- 40                   RTI
```

Es wird auf die Bank zurückgeschaltet, die vor dem IRQ an war.
Angesprungen wird diese Routine vom Ende des IRQ-Handlers im MLI ($D13A) nach erfolgreichem Service. Für „Monitor-IRQs" führt der RTI ins laufende Programm zurück. Passiert der IRQ dagegen „in der Language Card", muß erst die Speicherstelle $0045 wieder in Ordnung gebracht werden. Der RTI führt in diesem Fall zu $BF50 (s. „Interrupts").

```
BFEB- AD 8B C0    INTENTRY: LDA  $C08B      ; „inoffiziell"
BFEE- AD 8B C0              LDA  $C08B
BFF1- 4C 3A D1              JMP  $C13A      ; zum IRQ-Handler im MLI
```

Hierher zeigt der IRQ-Vektor auf der Seite 3 ($03FE/FF). Bei einem „Monitor-IRQ" wurde nur der Akku-Inhalt auf $0045 gespeichert, bevor hierher gesprungen wird. Für den Fall, daß die Language Card während des IRQ an war, wurden zuerst INTBANK1 und INTBANK2 gesetzt sowie die Speicherstelle $0045 gerettet ($FF9E).

```
BFF4- **          BANKSEL:  DS  01         ; „inoffiziell"
BFF5- **          BANKSEL1: DS  01
```

In diesen beiden Speicherstellen landen die IDs der momentan aktiven Bank (1, 2 oder ROM) bei einem MLI-Aufruf ($BFB7). Bei MLIEXIT ($BFA0) wird hiermit bestimmt, welche Bank vor dem MLI-Aufruf an war und entsprechend zurückgeschaltet.

```
BFF6- 4C 8B C0    SYSDEATH2: LDA  $C08B     ; „inoffiziell"
BFF9- 4C E4 D1               JMP  $D1E4     ; SYSTEM DEATH ohne ERR#
```

Dieses Stück Code ist etwas merkwürdig: Der REBOOT-Vektor ($BF09) zeigt vor der Installation von REBOOT in der Bank 2 der Language Card hierher und wird auf den REBOOT-Handler geändert, sobald die Kopieraktion von REBOOT erfolgt ist. Da diese

Dinge im Relocator passieren, d.h. bevor ProDOS überhaupt installiert ist, ist es unmöglich, daß ein Aufruf von REBOOT ein solches Ende nimmt.

```
BFFC- **          IBAKVER:   DS  01          ; „offiziell" + WRITE
BFFD- **          IVERSION:  DS  01
```

Diese beiden Bytes sollten laut Apple, Inc. von jedem System-Programm gesetzt werden. $BFFC steht für die früheste Version des MLI, die vom System-Programm benötigt wird, $BFFD für die Versionsnummer des System-Programms selber.

```
BFFE- 00          KBAKVER:   DFB  00          ; „offiziell"
BFFF- 00          KVERSION:  DFB  00
```

Diese letzten beiden Speicherstellen stehen für das MLI selber, $BFFE steht für „mindestens kompatibel mit Version xx", also „kann Files ab Version xx verarbeiten", auf $BFFF steht die Versionsnummer selber.

Die einzige dieser vier Speicherstellen, die momentan im MLI verwendet wird, ist $BFFF, also die Nummer der momentan installierten MLI-Version. Sie wird bei einem OPEN mit der Versionsnummer des Files verglichen ($E118). Ist die Versionsnummer des Files höher als die auf $BFFF verzeichnete Version des MLI, folgt ERR#4E: INCOMPATIBLE FILE FORMAT und Abbruch. In der Version 1.1.1 ist dieser Test allerdings nicht mehr vorhanden.

# 4.2. Unterschiede zu 48k-Systemen

Dieses Kapitel könnte eigentlich recht kurz ausfallen: die ProDOS-Versionen 1.0.1 und 1.0.2 funktionieren für 48k **nicht**, d.h. der ganze Aufwand mit Relocator und vorbereiteten System Global Pages ist bei diesen beiden Versionen für die Katz. Apple, Inc. verkauft momentan ein System, das entgegen den Beschreibungen im „Technical Manual" nur auf 64k bzw. „128k"-System läuft.

Begründung:
Das gesamte MLI liegt im File PRODOS in einer auf „64k" lauffähigen Form vor und wird in diesem Fall nur kopiert. Für „48k" muß zusätzlich relokalisiert werden.
Der Relocator behandelt alle absoluten Adressierungen korrekt. Ein direktes Setzen von Zeropage-Pointern kann mit dieser Methode natürlich nicht erkannt und korrigiert werden, also z.B:

```
    LDA  #$00
    STA  $48  ; Lobyte
    LDA  #$F6
    STA  $49  ; Hibyte, $48/$49 zeigen jetzt auf $F600.
```

Folglicherweise geschieht dieses Setzen im MLI auch nicht durch einen direkten Load (immediate Adressing), sondern durch das Laden aus einer (vorher vom Relocator korrigierten) absoluten Adressierung. Beispiel:

```
DDE5-  AD 2A D8     LDA  $D82A
       .....
DDEA-  85 45        STA  $45        ; Setzen des RWTB-Pointers (Adresse hi)
```

Auf $D828 findet sich eine absolute Adressierung, die beim Boot durch den Relocator korrigiert wurde:

```
D828-  AD 02 F6     LDA  $F602
```

An zwei Stellen hat der Programmierer von ProDOS diese Methode leider nicht angewendet, nämlich beim Anhängen eines Subdirectory-Blocks in CREATE sowie innerhalb von ENTRY & UPDATE bzw. FLUSH. Hier wird der entsprechende Pointer direkt gesetzt und bleibt auf $F600 (anstatt auf $B600 gesetzt zu werden).

Die entsprechende Stelle in CREATE:

```
D57D-  A9 F6        LDA  #$F6
D57F-  85 49        STA  $49
D581-  A9 04        LDA  #$04       ; $48/$49 = $F604: Directory-Startadresse
```

Hier soll die Basisadresse des File-Eintrags der soeben um einen Block erweiterten Subdirectory auf den neuesten Stand gebracht werden (BLOCKS USED/EOF). Für ein 48k-System wird dann zum „Directory-Eintrag" innerhalb des APPLESOFT-ROMS ein weiterer Block dazuaddiert.

Die zweite Stelle ist eine Subroutine innerhalb von ENTRY & UPDATE, die auch von FLUSH her aufgerufen wird. Hier wird ebenfalls die Startadresse eines File-Eintrags aus „Directory-Start + Eintragsnummer * Eintragslänge" berechnet:

```
D692-  A9 F6        LDA  #$F6
D694-  85 49        STA  $49
D696-  A9 04        LDA  #$04       ; $48/$49 = $F604: Directory-Startadresse
```

Da FIND FILE ENTRY eine andere Multiplikationsroutine benutzt, funktionieren alle anderen MLI-Befehle (OPEN, GET FILE INFO, READ etc.) einwandfrei – solange, bis das File CLOSEd wird.
Dieser Fehler läßt sich mit dem EXER.SYSTEM nachweisen – der Relocator des EXER.SYSTEM arbeitet auch bei 48k einwandfrei und der EXER meldet sich dann auch folgerichtig bei einem CLOSE oder SET FILE INFO mit ERR#51: DIRECTORY STRUCTURE DAMAGED.

Die folgende Beschreibung der System Global Page und der restlichen Konfiguration für 48k-Systeme erfolgt damit in der frohen Erwartung, daß das irgendwann einmal jemand bei Apple, Inc. merkt.

Innerhalb des Boot/Relocators wird auf „64k" getestet ($202E: Aufruf von $23B0). Hat die Maschine nur 48k RAM, werden die Startadressen der Parameterlisten für den MOVER auf die „48k"-Parameter gesetzt.

Nach der Installation ergibt sich folgende Speicherbelegung durch das MLI:

$9000..$AFFF: MLI
$B000..$B0FF: Variablenbereich
$B100..$B1FF: PREFIX und Pathname, SET TIME auf $B142
$B200..$B2FF: VCB
$B300..$B3FF: FCB
$B400..$B5FF: VBM Space
$B600..$B7FF: Directory Space
$B800..$BEFF: RWTB für die DISKII
$BF00..$BFFF: System Global Page
Die RAMDisk-RWTB (64k: $FF00) und der IRQ/RESET-Handler (64k: $FF80..$FFFF) entfallen.

Die Routine REBOOT, die normalerweise in die Bank 2 der Language Card kopiert wird, entfällt ebenfalls.

Innerhalb des MLI wurden alle absoluten Referenzen um -$4000 korrigiert, ansonsten werden keine Änderungen vorgenommen. Der einzige Teil von ProDOS, in dem sich unterschiedlicher Code befindet, ist die System Global Page.

```
BF00-  4C 00 90    MLIENTRY:  JMP  $9000
```

Der Sprung zu $BFB7 entfällt, Bank-switching und das Setzen von MLIACTIVE finden nicht statt.

```
BF03-  4C D5 90    REBOOT:    JMP  $90D5
```

Da REBOOT nicht kopiert wurde, passiert hier überhaupt nichts: auf $90D5 steht ein RTS. Nach Überspringen des COMMAND-Bytes und der Parameterblockadresse kehrt das Programm ohne jede Fehlermeldung zurück und versucht, im „user"-Programm weiterzumachen.

```
BF06-  4C B1 42    SETTIME:   JMP  $B142
```

Die Uhrenroutine wurde auf $B142 relokalisiert und dürfte normal funktionieren.

```
BF09-  4C DA 91    SET-SYSERR: JMP  $91DA
```

SET SYSERR steht innerhalb des MLI und wurde relokalisiert.

```
BF0C-  4C E6 91    SYSDEATH:  JMP  $91E6
```

SYSTEM DEATH steht innerhalb des MLI und wurde relokalisiert.

```
BF0F-  **          SYSERR:    DS  01          ; keine Änderung

BF10..             DRV1TBL:   DW  $90A2
BF20..             DRV2TBL:   DW  $90A2
```

Die NO DEVICE-Vektoren zeigen auf $90A2, wo ERR#28 gesetzt wird. Diese Routine steht innerhalb des MLI und wurde korrekt relokalisiert. Der entsprechende Vektor zur DISKII-RWTB zeigt für 48k-Systeme auf $B800.

```
BF30- **          LASTDEV:   DS  01          ; keine Änderung

BF31- **          DEVNUM:    DS  01          ; keine Änderung

BF32..BF3F        DEVTBL:    DFB  $00,00...  ; keine Änderung


BF40-43 4F ...    COPR:      ASC 'COPYRIGHT APPLE  1983  '
```

Das Copyright ist hier etwas „ausführlicher" und belegt die Speicherstellen $BF50..$BF57 (LANGIRQ, INTACCU, INTBANK2) mit.

```
BF58- C0          SBITMAP:   DFB  $C0         ; Seiten $0000 und $0100 belegt
BF59- 00 00 00 ....
BF6A- FF FF FF               DFB  $FF,FF,FF
BF6D- FF FF FF               DFB  $FF,FF,FF
```

In der SYSTEM BIT MAP sind nicht nur die Speicherseiten $0000..$0100 und $BF00 belegt, sondern der Bereich $9000..$BEFF ebenfalls.

```
BF70..BF7F        BUFTBL:    DFB  $00,...    ; keine Änderung

BF80..BF87        INTRUPTS:  DFB  $00,...    ; keine Änderung

BF88- **          INTAREG:   DS  01          ; keine Änderung
BF89- **          INTXREG:   DS  01          ; keine Änderung

BF8A- **          INTYREG:   DS  01          ; keine Änderung
BF8B- **          INTSREG:   DS  01          ; keine Änderung
BF8C- **          INTPREG:   DS  01          ; keine Änderung

BF8D- 02          INTBANK1:  DFB  $02
```

INTBANK1 ist bereits in der vordefinierten 48k-SGP im File PRODOS auf $02 gesetzt. Die Routine INTEXIT ($BFD0), die auch auf dieser SGP unverändert existiert, führt damit überhaupt kein Bank-switching aus und INTBANK1 bleibt grundsätzlich unverändert – was allerdings bei dem Wert $01 auch der Fall wäre.

```
BF8D- ** **       INTADDR:   DS  02          ; keine Änderung

BF90- ** **       DATE:      DS  02          ; keine Änderung

BF92- ** **       TIME:      DS  02          ; keine Änderung

BF94- **          LEVEL:     DS  01          ; keine Änderung
BF95- **          BUBIT:     DS  01          ; keine Änderung

BF96- ** **       SPARE:     DS  02          ; keine Änderung

BF98- **          MACHID:    DS  01          ; keine Änderung
BF99- **          ROMBIT:    DS  01          ; keine Änderung

BF9A- **          PREFIX:    DS  01          ; keine Änderung

BF9B- 00          MLIACTIVE: DFB  00
```

In MLIAKTIVE wird zwar nach wie vor beim internen Exit des MLI ($9078) auf Bit 7 ein Nuller hineingeshiftet, da aber bei $BF00 direkt nach $9000 gesprungen wird, ohne MLIAKTIVE zu verändern, bleibt diese Speicherstelle ständig auf Null.

```
BF9C-  ** **      MLIRTS:    DS  02          ; keine Änderung

BF9E-  **         XSAVE:     DS  02          ; keine Änderung
BF9F-  **         YSAVE:     DS  02          ; keine Änderung


BFA0-  68         MLIEXIT:   PLA
BFA1-  40                    RTI
```

Das Zurückschalten auf die Bank, die vor dem MLI-Aufruf aktiv war, entfällt. Das beim internen MLI-Exit geladene BANKID wird ignoriert.

```
BFA3..BFB6                   DFB  $00,00,00,00...

BFB7..BFBF                   DFB  $00,00,00,00...
```

MLIENT1, das Umschalten auf Bank 1 und Speichern des BANKID der vorherigen Bank vor dem Sprung ins MLI entfällt, das MLI wird von $BF00 direkt angesprungen.

```
BFD0-  AD 8B BF   INTEXIT:   LDA  $BF8D      ; INTBANK1 - hier immer Wert 02
BFD3-  F0 0D                 BEQ  $BFE2      ; hier ein „never"
BFD5-  30 08                 BMI  $BFDF      ; noch ein „never"
BFD7-  4A                    LSR             ; dieser Befehl bekommt hier einen

BFD8-  90 0D                 BCC $BFE7         Sinn: kein Bank-switching      ...
BFE7-  AD 88 BF              LDA  $BF88      ; INTAREG - Reload des Akku
BFEA-  40                    RTI
```

Die Routine INTEXIT steht hier in derselben Form wie für 64k – das „LSR" und der anschließende Return zum durch den IRQ unterbrochenen Programm wird hier genutzt. INTBANK1 mit dem Inhalt $01 würde allerdings zum selben Ergebnis führen. Der Rücksprung ins laufende Programm findet hier ohne jeden Test auf „RWTB-Pointer verändert" statt – hierfür hätte man MLIACTIVE nutzen können (wenn nicht die Routine, die dieses Flag setzt, ebenfalls entfallen wäre).

```
BFEB- 4C 3A 91    INTENTRY:  JMP  $913A
BFEE- 00 ...                 DFB  $00,00,...
```

Bei INTENTRY wird nicht mehr auf die Bank 1 der Language Card umgeschaltet, sondern direkt zur IRQ-Routine innerhalb des MLI gesprungen.

```
BFF4- 00          BANKSEL:   DFB  $00
BFF5- 00          BANKSEL1:  DFB  $00
```

Die Speicherstellen BANKSEL werden nicht mehr benutzt, weil eine Rückschaltung auf die vorherige Bank nicht nötig ist.

```
BFF6- 48 55 53..  SYSDEATH2: ASC  'HUSTON'
```

Anstelle des (auch bei 64k nicht benutzen) Vektors zu SYSTEM DEATH hat sich hier wieder einmal der Name HUSTON eingeschlichen

```
BFFC- **          IBAKVER:   DS  01          ; keine Änderung

BFFD- **          IVERSION:  DS  01          ; keine Änderung
BFFE- 00          KBAKVER:   DFB  $00        ; keine Änderung
BFFF- 00          KVERSION:  DFB  $00        ; keine Änderung
```

# Teil V – die RWTB der DISK II

## 5.1. Die Organisation einer Diskette – Tracks und Sektoren

Etwas Terminologie zum Anfang:
Hier und im weiteren bezieht sich der Begriff „Diskette" oder „Disk" auf die flache Kunststoffscheibe, die sich im Inneren des „Jackets" (Umschlag?) befindet. Diese Scheibe wird durch den Motor des Drives (Diskettenlaufwerk) zur Rotation gebracht.
Eine Diskette enthält an ihrer Oberfläche eine magnetisierbare Schicht. Diese Schicht ist in keiner Weise unterteilt, für die Aufteilung ist der Computer bzw. das angewendete Formatierprogramm verantwortlich.

**A) Aufteilung in Tracks**

Eine Diskette wird von jedem (Standard)-Betriebssystem auf dem APPLE in 35 Tracks (Spuren) unterteilt. Per Definition ist der Track 00 der äußerste, der Track mit der höchsten Nummer der innerste auf einer Diskette. Die Tracks verlaufen kreisförmig, (konzentrisch), also nicht spiralig, wie etwa bei einer Schallplatte.
Das Diskettenlaufwerk enthält einen Arm, an dem der Schreib/Lesekopf für die Diskette befestigt ist. Der Arm wird durch einen Schrittmotor von einem zum nächsten Track bewegt. Der Abstand zweier Tracks ist (normalerweise) durch die Auflösung des Schrittmotors bestimmt, d.h. durch die Strecke, die der Schrittmotor mit einem Schritt zurücklegt. Beim APPLE werden allerdings von einem Track zum nächsten zwei Schritte des Motors verwendet, um ein „Übersprechen" zwischen zwei Tracks zu vermeiden: der Schreib/Lesekopf bringt die Daten zweier zu dicht aneinanderliegender Tracks sonst durcheinander.
Es erfolgt keine Rückmeldung vom Schrittmotor aus zum Computer, d.h. der Computer muß selbst bestimmen, in welcher Position sich der Arm befindet.

Beim Formatieren tappt der Computer quasi im Dunklen – der Arm wird um eine hohe Anzahl von Tracks (hier: 40) nach außen bewegt, um sicherzustellen, daß er sich über Track 00 befindet.
Der Track wird „blind" geschrieben und verifiziert, danach wird der Arm um einen Track nach innen bewegt. Der Prozeß wiederholt sich beim APPLE 35 Mal. Danach sollte der Arm über Track 34 stehen und die eigentliche Formatierung ist damit beendet.

**B)Aufteilung in Sektoren**

Die Größe eines Sektors legt die kleinste Datenmenge fest, die vom Computer auf einmal gelesen/geschrieben werden kann.
Es wäre theoretisch denkbar, daß ein Track nur zwei Markierungen enthält:

„Hier fangen die Daten an" (Header)
„Hier hören die Daten auf" (Trailer)

Praktisch stößt das auf Schwierigkeiten:
Um ein einzige Byte von der Diskette zu holen, müßten dabei rund 4KByte gelesen, d.h. abgesucht werden, was entsprechend lange dauert. Viel unerfreulicher wirkt sich allerdings die Tatsache aus, daß für das Schreiben eines Bytes ebenfalls 4KByte geschrieben werden müßten – für diese 4 KByte muß ein entsprechender Speicherplatz innerhalb des Computers bereitgestellt sein.

Aus diesem Grund ist beim APPLE (und bei den meisten anderen Computern) ein Track in 16 Sektoren mit jeweils 256 Byte an Daten unterteilt.

Um einen Track in Sektoren zu unterteilen, gibt es zwei Möglichkeiten. Vorher allerdings wieder etwas Terminologie:
Der „Umschlag" einer Diskette enthält nahe dem (großen) Mittelloch noch ein zweites kleines Loch, durch das normalerweise ein Stück der Diskettenoberfläche sichtbar ist. Dreht man die Diskette (vorsichtig!) ein Stück „zu Fuß", stellt man fest, daß irgendwann unter dieser Aussparung ein Loch im Diskettenmaterial selber auftaucht, daß sogenannte Indexloch.
Bei der Drehung der Diskette kann der Zeitpunkt, an dem dieses Indexloch auftaucht, elektronisch bestimmt werden (Lichtschranke).

1) „Hart-Sektoriert"
Eine Form von Disketten, die heutzutage praktisch nicht mehr verwendet wird. Die Diskette enthält eine Anzahl von Indexlöchern, jedes Indexloch bestimmt den Anfang eines Sektors. Der Anfang jedes Sektors ist damit physikalisch festgelegt, die Anzahl der möglichen Sektoren auf einem Track allerdings auch.

2) „Soft-Sektoriert"
Die Diskette enthält nur ein Indexloch, das die physikalische Position jedes ersten Sektors (Sektor 0) auf einem Track festlegt. Damit ist die Anzahl der Sektoren pro Track (und damit die Länge eines einzelnen Sektors) beliebig, allerdings muß eine Zusatzinformation auf der Diskette geschaffen werden, die die einzelnen Sektoranfänge kennzeichnet.

Der APPLE verwendet soft-sektorierte Disketten, das Indexloch wird dabei auch noch ignoriert, d.h. die physikalische Position eines Sektors auf einem Track ist völlig undefiniert.

Jeder Sektor ist in zwei Teile unterteilt:

– das Address Field („Wo bin ich?" für den Computer);
– das Data Field (die Daten selber, sie folgen dem Address Field nach).

Das Address Field enthält folgende Informationen:

```
– D5 AA 96      ; „Address Field Header": hier fängt ein Address Field an
– xx yy         ; „Volume-Nummer" – wird von ProDOS ignoriert
– xx yy         ; Tracknummer – dieses Address Field befindet sich auf Track xx
– xx yy         ; Sektornummer – der nachfolgende Datensektor hat die Nummer yy
– zz zz         ; „Checksum" zur Überprüfung der Address-Daten
– DE AA EB      ; „Address Field Trailer": hier hört das Field auf (Überprüfung)
```

Das Address Field jedes Sektors wird nur beim Formatieren geschrieben – innerhalb des „Normalbetriebs" wird es nur noch gelesen. Die RWTB von ProDOS enthält z.B. überhaupt keine Routine, die Address Fields schreibt, diese Routine ist nur im FILER enthalten.

Das Datafield enthält folgendes:

```
– D5 AA AD      ; „Data Field Header": hier fängt ein Data Field an
 ....           ; die Daten selber
– zz            ; „Checksum" zur Überprüfung der Daten
– DE AA EB      ; „Data Field Trailer": hier hört das Field auf (Überprüfung)
```

Das Data Field kann im „Normalbetrieb" sowohl gelesen als auch beschrieben werden.

Verfolgen wir einen typischen Suchprozeß nach einem Sektor:

1. Der Motor des Laufwerks wird angeschaltet und READ gesetzt.
2. Das nächste vorbeikommende Address Field wird gelesen.
3. Der Computer weiß jetzt, über welchem Track sich der Arm befindet. Der Abstand zum Ziel-Track wird berechnet, der Arm wird entsprechend bewegt.
4. Auf dem Ziel-Track werden solange Address Fields gelesen, bis die Information „jetzt kommt Sektor xx" mit dem „Sektor gewünscht" übereinstimmt.

### C) „Gaps" und Geschwindigkeitsprobleme

Die tatsächliche Geschwindigkeit, mit der sich eine Diskette im Laufwerk dreht, hängt von vielen Faktoren ab (Reibung der Diskette im Jacket etc.), d.h. sie ist nicht 100%ig bestimmbar.
Normalerweise dauert eine Umdrehung der Diskette 0,2 Sekunden, Abweichungen nach oben oder unten liegen dabei im Bereich von rund 0.5 %.
Die Zeit, in der ein Bit geschrieben/gelesen wird, beträgt 4 Mikrosekunden.
Damit ist offensichtlich, daß eine Diskette, auf der ein Track lückenlos geschrieben wurde, zwar gelesen, nicht aber neu beschrieben werden kann – damit sich nach einer Umdrehung Track-Anfang und Track-Ende lückenlos aneinanderschließen, dürfte die Umdrehungsge-

schwindigkeit innerhalb von 0.2 Sekunden (eine Umdrehung) um weniger als 4 Mikrosekunden (1 Bit) abweichen, d.h. um weniger als 0,0002 %.
Aus diesem Grund ist zwischen „Anfang" und „Ende" eines Tracks eine Lücke („Gap 1") gesetzt:

Sektor $0F(Daten) ---- Gap 1 --- Sektor $00 (Address Field)

Dieses „Gap" wird folgendermaßen erzeugt:
Beim Formatieren wird zuerst eine Anzahl von „Autosync"-Bytes ($FF – siehe „Hardware Details") geschrieben, danach Sektor 0, Sektor 1 ... Sektor $0F. Der letzte Teil von Sektor $0F überschreibt einen Teil des Gap 1 mit Sicherheit – wieviel von diesem Gap übrig bleibt, ist eine Frage der Laufwerksgeschwindigkeit und unwichtig. Damit wird sichergestellt, daß bei einem (Neu)schreiben von Sektor $0F das physikalisch nachfolgende Address Field von Sektor 00 desselben Tracks nicht beschädigt wird.
Das Gap 1 ist normalerweise (bei justierter Motorgeschwindigkeit) rund $30 Bytes lang, bei Disketten, die mit dem FILER formatiert sind, ist es etwas kürzer (s. „Spezielles").

Beim Schreiben eines Data Field wären die Anforderungen an die Umdrehungsgeschwindigkeit ähnlich hoch. Aus diesem Grund existieren Gap 2 und Gap 3:

Sektor X0 (Address Field) -- Gap 2 -- Data Field -- Gap 3 -- Sektor X1 (Addr.)

Jedes Data Field „schwimmt" innerhalb eines bestimmten Raumes – Gap 2 ist normalerweise rund 8 Byte lang, Gap 3 rund 18 Byte.

Gap 3 hat die gleiche Funktion innerhalb des Tracks wie Gap 1 für den gesamten Track, nämlich den Ausgleich von Geschwindigkeitsdifferenzen.
Gap 2 hat eine Doppelfunktion: zum einen braucht der Prozessor etwas Zeit, um die im Address Field gelesenen Daten auszuwerten, zum anderen ist es beim Schreiben eines Data Field dadurch nicht nötig, auf 4 Mikrosekunden genau mit der eigentlichen Schreibaktion anzufangen. Das Programm fängt irgendwo innerhalb von Gap 2 an zu schreiben, schreibt das Data Field und beendet diese Aktion irgendwo in Gap 3.

## 5.2. Die logische Einteilung in Blocks – „Sektor-Skew"

Unter ProDOS umfaßt die kleinste Datenmenge, die gelesen/geschrieben werden kann, einen Block mit 512 Byte Inhalt.
Dieser Block besteht auf der Diskette aus zwei Sektoren mit jeweils 256 Byte, d.h. für das Lesen/Schreiben eines Blocks erfolgen jeweils zwei Diskettenzugriffe.
Die Blocks sind auf einer Standard-Diskette von 0 bis 279 ($117) durchnummeriert. Da ein Track 16 Sektoren enthält, gehen somit 8 Blocks auf einen Track.
Der Track 00 enthält dabei die Blocks 00-07, der Track 01 die Blocks 08-15 etc.

Um den Begriff und die Notwendigkeit des „Sektor-Skew" zu erklären, müssen wir kurz auf andere Betriebssysteme zurückgreifen. Hier sieht der Prozeß der Datenübergabe in etwa so aus:

1. Ein Sektor wird in einen DOS-eigenen Puffer gelesen (und dekodiert).
2. Die Daten werden in das Zielgebiet kopiert.
3. Wenn weitere Sektoren benötigt werden, zurück zu 1.

Während die Daten eines Sektors verarbeitet werden bzw. Vorbereitungen zum Lesen/Schreiben des nächsten Sektors getroffen werden, „rauscht" der physikalisch nächste Sektor am Schreib/Lesekopf (zumindest teilweise) vorbei. Die Folge: Wird danach ein zweiter Sektor benötigt, der physikalisch auf der Diskette unmittelbar dem ersten folgt, müßte das DOS fast eine volle Umdrehung der Diskette abwarten, bis dieser Sektor wieder vorbeikommt. Bei einem Hires-Bild von 8 kByte Länge würde sich diese Zeit bereits auf 6.4 Sekunden summieren.
Um diese Wartezeiten abzukürzen, wird eine DOS-interne Übersetzung von „logischer Sektor" zu „physikalischer Sektor" (Skewing) vorgenommen.
Für ProDOS findet diese Übersetzung sowohl auf Sektor- als auch auf Blockebene statt.
Ein Beispiel für das Lesen der Blocks 0-7:

Block 00: Sektoren 00 und 02
Block 01: Sektoren 04 und 06
Block 02: Sektoren 08 und 0A
Block 03: Sektoren 0C und 0E ; von Block 03 zu 04 werden
Block 04: Sektoren 01 und 03 ; 2 Sektoren übersprungen
Block 05: Sektoren 05 und 07
Block 06: Sektoren 09 und 0B
Block 07: Sektoren 0D und 0F

Auf dem nächsten Track geht es mit den Blocks 08-15 genauso weiter. Damit wird erreicht, daß die RWTB nach dem Lesen/Schreiben jedes Sektors Zeit für interne Operationen hat, ohne daß dabei der „nächste" Sektor für diese Umdrehung der Diskette verloren geht.

Der logische Ablauf ist für das Betriebssystem bzw. die Routine, die die entsprechenden Blocks anfordert, nach wie vor derselbe. Diese Übersetzung findet auf der untersten Ebene, nämlich innerhalb der RWTB statt.

Die Frage, warum man nicht einfach beim Formatieren die einzelnen Sektoren bzw. deren Address Fields gleich in der nötigen Reihenfolge beschrieben hat, ist mehr oder weniger akademisch, vom Ablauf her kommt es auf dasselbe heraus.
In den Versionen von DOS vor 3.3 war dieser Prozeß auch tatsächlich „ausgelagert", d.h. das Formatierprogramm hat den Sektor-Skew in die Address Fields hineingeschrieben, eine weitere Umsetzung fand demzufolge im „Normalbetrieb" nicht mehr statt.

Eine Gegenüberstellung des Sektor-Skews der vier bekannten Betriebssysteme für den APPLE:

| Physikalisch | ProDOS | UCSD-Pascal | DOS3.3 | CP/M |
|---|---|---|---|---|
| 0 | Block0L | Block0L | 0 | Records 01/02 |
| 1 | Block4L | Block4L | 7 | Records 23/24 |
| 2 | Block0H | Block0H | E | Records 13/14 |
| 3 | Block4H | Block4H | 6 | Records 03/04 |
| 4 | Block1L | Block1L | D | Records 25/26 |
| 5 | Block5L | Block5L | 5 | Records 15/16 |
| 6 | Block1H | Block1H | C | Records 05/06 |
| 7 | Block5H | Block5H | 4 | Records 27/28 |
| 8 | Block2L | Block2L | B | Records 17/18 |
| 9 | Block6L | Block6L | 3 | Records 07/08 |
| A | Block2H | Block2H | A | Records 29/30 |
| B | Block6H | Block6H | 2 | Records 19/20 |
| C | Block3L | Block3L | 9 | Records 09/10 |
| D | Block7L | Block7L | 1 | Records 31/32 |
| E | Block3H | Block3H | 8 | Records 21/22 |
| F | Block7H | Block7H | F | Records 11/12 |

Wie man sieht, ist das Skewing für ProDOS und Pascal identisch. Es wird als „2-ascending Skew" bezeichnet.

Der Vollständigkeit halber soll hier auch noch etwas Licht in das Skewing von DOS3.3 und CP/M gebracht werden:

Auch bei CP/M wird in aufsteigender Reihenfolge gelesen, also:

Records 01/02: Sektor 00
Records 03/04: Sektor 03
Records 05/06: Sektor 06
Records 07/08: Sektor 09 etc.

Der größere „Abstand" bei der Umsetzung („3-ascending skew") kommt daher, weil hier immer nur 128-Byte („Record") auf einmal vom DOS in das Zielgebiet kopiert werden. Das Programm, das die Daten von der Disk angefordert hat, muß also pro Sektor zweimal entscheiden, ob es bereits genug hat. Zusätzlich müssen Pointer etc. zweimal pro Sektor eingerichtet werden, und das dauert unter Umständen eben etwas länger.

DOS 3.3 nimmt eine Sonderstellung ein, hier wird mit einem absteigenden („descending") Skew gearbeitet. DOS liest die Sektoren innerhalb eines Tracks immer von „hinten", also:

Sektor $0F: -> Sektor $0F
Sektor $0E: -> Sektor $02
Sektor $0D: -> Sektor $04
Sektor $0C: -> Sektor $06
Sektor $0B: -> Sektor $08 etc.

Dieser Skew wird als „2-descending" bezeichnet.

# 5.3. „Nibbilizing"/„Denibbilizing" – Lesen und Schreiben eines Sektors

Das Lesen eines 256-Byte Sektors von der Diskette wäre am schönsten in folgender Form:

```
GET-SECTOR:     LDY  #$00      ; Zähler für 256 Byte
GET-BYTE:       LDA  DISK-IN   ; genaueres dazu s. „Hardware Details"
                STA  TARGET,Y
                INY
                BNE  GET-BYTE
```

Daß dieses Kapitel dennoch etwas länger ausfällt, liegt natürlich daran, daß die Geschichte leider so nicht funktioniert.

**A) Voraussetzungen für „Disk"bytes**

Ein „Disk"byte muß aus technischen Gründen zwei Voraussetzungen erfüllen:

1) Das höchste Bit (Bit 7) jedes Bytes muß gesetzt sein. Dieses Bit wird als erstes jedes Bytes gelesen/geschrieben. Das Warum ist in „Hardware Details" ausführlich erläutert.

2) Ein Byte darf maximal ein Paar aufeinanderfolgende „Nuller" haben. Der Grund hierfür liegt im „Datenseparator" der Controller-Elektronik. Die Erklärung führt ziemlich weit in die Elektronik hinein und wir begnügen uns hier mit der Feststellung, daß dem so ist.

Die erste Forderung reduziert die Anzahl der „brauchbaren" Bytes auf den Bereich $80-$FF, d.h. von 256 auf 128 Möglichkeiten.

Zusammen mit der zweiten Forderung bleiben folgende Werte übrig:

```
0000- 96  97  9A  9B  9D  9E  9F  A6  A7  AB  AC  AD  AE  AF  B2  B3
0010- B4  B5  B6  B7  B9  BA  BB  BC  BD  BE  BF  CB  CD  CE  CF  D3
0020- D6  D7  D9  DA  DB  DC  DD  DE  DF  E5  E6  E7  E9  EA  EB  EC
0030- ED  EE  EF  F2  F3  F4  F5  F6  F7  F9  FA  FB  FC  FD  FE  FF
```

Nach der Definition sind noch fünf weitere Werte möglich:

1010 0101 (A5)
1010 1001 (A9)
1010 1010 (AA)
1100 1010 (CA)
1101 0101 (D5)

Diese fünf Bytes sind die einzigen, die im Bereich Bit 6-0 keine zwei aufeinanderfolgenden „Einser" haben. Zwei davon ($AA/$D5) werden als „reserviert" erklärt, d.h. sie tauchen nie

als Daten auf, die restlichen 3 werden einfach nicht benutzt. Damit ist der Bereich an möglichen „Disk"bytes auf 64 Werte eingegrenzt.

**B) „EVEN-ODD"**

Es gibt eine sehr einfache Methode, die gestellten Forderungen zu erfüllen. Das zu schreibende Byte erhält die Form xyxy xyxy und wird in zwei Teile zerlegt:

1x1x 1x1x und 1y1y 1y1y

Das ist recht leicht in ein Programm umzusetzen:

```
DOUBLE-BYTE:    LDA  #BYTE       ; xyxy xyxy
                LSR  A           ; 0xyx yxyx
                ORA  #$AA        ; 1010 1010
                JSR  DISKOUT =>    1x1x 1x1x
                LDA  #BYTE
                ORA  #$AA        ; 1010 1010
                JSR  DISKOUT =>    1y1y 1y1y
```

Damit sind die gestellten Forderungen mit Sicherheit erfüllt. Ein solches Byte läßt sich auch sehr einfach wieder „zusammensetzen":

```
GET-DOUBLE:     LDA  DISKIN      ; erste Hälfte: 1x1x 1x1x
                SEC
                ROL  A           ; aus 1x1x 1x1x wird x1x1 x1x1
                STA  TEMP
                LDA  DISKIN      ; zweite Hälfte: 1y1y 1y1y
                AND  TEMP        ; => xyxy xyxy
```

Diese Methode wurde früher auch angewendet. Sie ist unkompliziert, hat aber einen gewaltigen Nachteil: für jedes Byte werden zwei Byte auf der Disk benötigt. Dementsprechend fällt dann das Fassungsvermögen einer normalen 5 1/4 Zoll-Disk aus: es sind rund 80 Kilobyte („single density").
Weil die Methode so unkompliziert ist, wird sie dennoch innerhalb der DiskII-RWTB da verwendet, wo es darauf ankommt, etwas schnell zu lesen, und wo die Datenmenge klein genug ist. Die Rede ist hier von den Address Fields. Da wir hier insgesamt 4 Byte Inhalt haben, fällt es nicht auf, daß für jedes Byte der doppelte Platz auf der Diskette „verbraten" wird.
Eine Routine, die diese „even-odd"-kodierten Bytes schreibt, wird man in der RWTB vergeblich suchen; sie wird nur zur Formatierung gebraucht und findet sich deshalb im FILER, wobei hier auch noch nicht dokumentierte Dinge geschehen (s. „Spezielles").
Die Routine, die diese Bytes liest, hat den Namen RDADDR und fängt auf $FBFD an.

**C) „6:2"**

Für größere Datenmengen haben sich die APPLE-Entwickler etwas wesentlich komplizierteres ausgedacht: um 256 mögliche Werte so zu „verpacken", daß sie sich mit 64 Werten ausdrücken lassen, wird eine Umsetzung vorgenommen.

Aus den Bytes A, B, C (hier in einzelnen Bits dargestellt)

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A7 | A6 | A5 | A4 | A3 | A2 | A1 | A0 | | A7 | A6 | A5 | A4 | A3 | A2 | ** ** |
| B7 | B6 | B5 | B4 | B3 | B2 | B1 | B0 | wird dabei | B7 | B6 | B5 | B4 | B3 | B2 | ** ** |
| C7 | C6 | C5 | C4 | C3 | C2 | C1 | C0 | | C7 | C6 | C5 | C4 | C3 | C2 | ** ** |
| | | | | | | | | | | | | | | | |
| und das „zusätzliche" Byte | | | | | | | | | C0 | C1 | B0 | B1 | A0 | A1 | ** ** |

Aus drei Bytes werden damit vier „Nibbles". Diese Bezeichnung ist eigentlich irreführend, weil ein Nibble ein Vier-Bit-Byte bezeichnet, aber sie hat sich nun einmal eingebürgert. Eingeführt wurde sie meines Wissens von Worth & Lechner.

Aus 256 Datenbytes ergeben sich damit 256 * 4/3 = 342 Bytes auf der Disk.

Ein Absatz für DOS3.3-Programmierer:
Die Prenibble-Routine in DOS3.3 arbeitet nach demselben Prinzip, die Umsetzung findet aber etwas anders statt. DOS kopiert die Bytes aus dem „user"-Puffer in den eigenen Nibble-Puffer($BB00-$BBFF), nimmt die beiden LSBs heraus (=> $BC00-$BC55) und shiftet dabei das ursprüngliche Byte zweimal nach rechts, d.h. im Puffer $BB00-$BBFF stehen danach Bytes im Wertebereich von $00-$3F (** ** A7 A6 A5 A4 A3 A2). Unter ProDOS bleiben die ursprünglichen Bytes stehen, sie werden nicht geshiftet. Vor dem Umsetzen in „Disk"bytes werden einfach die beiden LSBs maskiert (AND #$FC). Der entstehende Datenstrom auf der Disk ist derselbe, deshalb können ProDOS-Disks auch via RWTS gelesen werden. Bei der Beschreibung des „6:2"-Nibble-Prozesses haben sich Worth & Lechner den einzigen Flüchtigkeitsfehler im gesamten „Beneath Apple DOS" geleistet: die Reihenfolge der Bits in den „zusätzlichen" Bytes ist auch für DOS3.3 x1 x0 und nicht x0 x1 wie angegeben (Kapitel 3, Seite 20).

Kommen wir zur praktischen Ausführung des „Prenibble"-Prozesses.
Um aus jeweils drei Bytes vier „Nibbles" zu erhalten, werden drei Basisadressen innerhalb der 256 Byte berechnet, d.h. der Puffer wird in drei gleiche Teile aufgespaltet.
Gehen wir von einem Puffer mit der Startadresse $1000 aus, dann sind die Basisadressen:

```
$1000 - Start auf den ersten Byte (Byte 00)    erstes Drittel
$1056 - Start auf dem 87.Byte                  zweites Drittel
$10AC - Start auf dem 173.Byte                 drittes Drittel
```

Diese Basisadressen werden auf $FDF0 gesetzt. Danach folgt eine Schleife ($FE1A):

```
        LDA  $1000,Y
        AND  #$03              ; nur Bit 0/1
        TAX
        LDA  $F9E0,X
```

Die Tabelle auf $F9E0 enthält die Werte $00/$08/$04/$0C, die vier möglichen Kombinationen sind damit:

00 => 0000 0000
01 => 0000 1000
10 => 0000 0100
11 => 0000 1100

Mit dem nächsten Drittel des Puffers ist die Umsetzung analog, nur enthält die dabei adressierte Tabelle ($F9C0) die Werte $00/$20/$10/$30:

00 => 0000 **00
01 => 0010 **00
10 => 0001 **00
11 => 0011 **00

Mit dem letzten Drittel des Puffers werden schließlich Bit 7/6 des „zusätzlichen" Bytes gefüllt (Umsetzungstabelle auf $F9A0 mit den Werten $00/$80/$40/$C0):

00 => 00** **00
01 => 10** **00
10 => 01** **00
11 => 11** **00

Das so berechnete Byte hat (für den ersten Durchlauf) damit die Form:

$AC_1\ AC_0\ 56_1\ 56_0\ 00_1\ 00_0$ (0 0)

Es wird auf $FB55 gespeichert, das durch den nächsten Durchlauf ($AD/$57/$01) errechnete Byte auf $FB54, bis $FB00 erreicht ist.
Die letzten drei Bytes im Puffer für die „zusätzlichen" Bytes:

FB02- $FF_1\ FF_0\ A9_1\ A9_0\ 53_1\ 53_0$ (0 0)
FB01- xx xx $AA_1\ AA_0\ 54_1\ 54_0$ (0 0)
FB00- xx xx $AB_1\ AB_0\ 55_1\ 55_0$ (0 0)

Daß Bit 7/6 der letzten beiden Bytes ungültig sind, liegt daran, daß sich 256 nicht ohne Rest durch 3 teilen läßt.

Zum Schreiben auf die Diskette müßte jetzt nur noch eine 1:1-Umsetzung vorgenommen werden, also z.B. mit einer Tabelle, die die erlaubten „Disk"bytes enthält. Da hier jeweils die letzten beiden Bits eines Bytes nicht mehr zu berücksichtigt werden brauchen, sind die Bytes innerhalb dieser Tabelle im Vierer-Abstand eingetragen. Ein Beispiel:

```
WRTRANS:
FA03- 96 ** ** ** 97 ** ** **  9A ** ** ** 9B ** ** **
FA13- 9D ** ** ** 9E ** ** **  9F ** ** ** A6 ** ** **
FA23- .....
```

```
WRITE-BYTE:     LDA  BUFFER,Y        ; ein Byte aus dem Puffer
                AND  #$FC            ; Bit 0/1 entfernen
                TAX
                LDA  WRTRANS,X
                STA  SHIFTREG        ; Schreiben auf die Disk
```

Das funktioniert sogar in der Praxis, die entsprechende Tabelle findet sich auf der oben angegebenen Adresse. Es fragt sich nur, wie lange es funktioniert – hier ist nicht die geringste Möglichkeit zur Entdeckung eines Fehlers beim Lesen dieser Daten enthalten.
Aus diesem Grund wird mit den zu schreibenden Bytes eine EXOR-Kette gebildet und die 342 Bytes werden in folgender Form auf die Diskette geschrieben:

```
   0 EXOR FB55 => WRTRANS  => Disk
FB55 EXOR FB54 => WRTRANS  => Disk
FB54 EXOR FB53 => WRTRANS  => Disk
     .....
FB02 EXOR FB01 => WRTRANS  => Disk
FB01 EXOR FB00 => WRTRANS  => Disk
FB00 EXOR 1000 =>      #$FC => WRTRANS  => Disk
1000 EXOR 1001 => AND #$FC => WRTRANS  => Disk
1001 EXOR 1002 => AND #$FC => WRTRANS  => Disk
     .....
10FE EXOR 10FF => AND #$FC => WRTRANS  => Disk
10FF ----------------------------------------> WRTRANS  => Disk
```

Bei diesem Beispiel wurde wieder von einem Puffer ausgegangen, der auf $1000 startet. Die „zusätzlichen" Bytes werden in absteigender Reihenfolge, der „eigentliche" Pufferinhalt in aufsteigender Reihenfolge geschrieben. Beim Schreiben des „eigentlichen" Pufferinhalts müssen die Bits 0 und 1 maskiert werden, bei den „zusätzlichen" Bytes sind diese Bits bei der Berechnung auf Null gesetzt, dieser Schritt entfällt damit.

Wer sich die Mühe des Nachzählens macht, wird feststellen, daß hier nicht 342, sondern insgesamt 343 Bytes geschrieben wurden. Um dieses 343.te Byte, die „Checksum", zu erklären, müssen wir uns allerdings vorher mit dem Lesen und „Denibbilizing" der geschriebenen Daten beschäftigen.

Das Lesen der Daten geschieht in umgekehrter Reihenfolge: Zuerst wird der Puffer für die „zusätzlichen" Bytes ($FB00-$FB55) von oben nach unten gefüllt, danach die Speicherseite im Zielgebiet in aufsteigender Reihenfolge.

Um dabei die Auflösung der EXOR-Kette und die Funktion der „Checksum" darzustellen, greifen wir zuerst einmal auf den Prozeß zurück, der unter DOS 3.3 abläuft und dasselbe Ergebnis zur Folge hat.
DOS 3.3 hat eine (einzige) Übersetzungstabelle für READ (RDTRANS). Der niedrigste Wert, der von der Diskette gelesen werden kann, ist $96. Folglicherweise fängt diese Tabelle auf $BA96 an und wird mit der Basis $BA00 adressiert. Die Tabelle enthält die umgekehrten Zuordnungen, d.h. wurde das Byte $00 bei WRITE mit $96 übersetzt, wird es bei READ auf

$00 zurück-übersetzt ($BA96 enthält den Wert $00). Die DOS-Tabelle umfaßt den Wertebereich $00-$3F.
Diese Tabelle (RDTRANS1) existiert auch für ProDOS. Sie hat hier die Basisadresse $F900 und denselben Zweck, aber einen anderen Inhalt: hier geht der Wertebereich von $00-$FF in Vierer-Schritten (Bit 0/1 sind ausgeklammert).

Ein Beispiel mit 3 Bytes, die den Wert $3A/$22/$10 haben, zuerst für WRITE (DOS und ProDOS-Prinzip):

```
0000 0000 ($00)
————————    EXOR = 0011 1010 ($3A) via WRTRANS => $FA zur Disk
0011 1010 ($3A)

0011 1010 ($3A)
————————    EXOR = 0001 1000 ($18) via WRTRANS => $BD zur Disk
0010 0010 ($22)

0010 0010 ($22)
————————    EXOR = 0011 0010 ($32) via WRTRANS => $EF zur Disk
0001 0000 ($10)
```

READ liest diese drei Byte ($FA/$BD/$EF) von der Diskette und verwandelt sie zurück (nur DOS 3.3):

```
0000 0000 ($00)
————————    EXOR = 0011 1010 ($3A)
$FA => RDTRANS 0011 1010 ($3A)

0011 1010 ($3A)
————————    EXOR = 0010 0010 ($22)
$BD => RDTRANS 0001 1000 ($18)

0010 0010 ($22)
————————    EXOR = 0001 0000 ($10)
$EF => RDTRANS 0011 0010 ($32)
```

Wie man hier sieht, wird beim Lesen das Ergebnis der vorherigen Verknüpfung immer weiter „mitgeschleppt". Das wirkt fast wie Zauberei (ist es vielleicht auch), aber es kommen „hinten" wieder die richtigen Werte heraus. Darüber kommen wir zur „Checksum":
Schreiben wir bei WRITE als viertes Byte den Wert $10 **ohne** vorheriges EXOR:
$10 via WRTRANS => $9D auf die Disk und hängen wir diesen Wert an das Lesebeispiel an:

```
„Ergebnis" von 1..3: 0001 0000 ($10)
————————    EXOR = 0000 0000 ($00)
$9D => RDTRANS 0001 0000$ (10)
```

Da sich die EXOR-Kette über den gesamten Sektorinhalt hinzieht, lassen sich damit sämtliche Bytes überprüfen. Das Lesen der „Checksum" muß zusammen mit der laufenden EXOR-Kette den Wert $00 ergeben.
Natürlich gibt es theoretisch auch hier noch Möglichkeiten, daß der Inhalt trotz „EXOR Checksum = $00" nicht stimmt – aber das ist etwas für Mathematiker: die Wahrscheinlichkeit liegt bei 1 zu 1 000 000 000 * 1 000 000 000 * (341! / 2!) * (8!), also jenseits von Gut und Böse.
Unter DOS 3.3 liegen nach diesem Prozeß 342 Bytes vor, die „zusätzlichen" Bytes müssen wieder auf auf die ursprünglichen Bytes (Bit 0/1) verteilt werden.

Die ProDOS-RWTB erledigt diesen Prozeß gleich während des Lesens mit, und das macht sie wirklich kompliziert:
Die „zusätzlichen" Bytes werden in absteigender Reihenfolge in den RWTB-eigenen Puffer ($FB00-$FB55) gelesen. Ein EXOR findet dabei nicht statt(!), die gelesenen Bytes werden lediglich via RDTRANS1 übersetzt und gespeichert. Vom Ergebnis her kommt das auf dasselbe heraus. Damit die darauf folgenden Daten überprüft werden können, muß die Checksum „mitlaufen", d.h. sie wird berechnet und zwischengespeichert, hat aber hier keinen Einfluß auf die gelesenen Daten. Dieser Teil der READ-Routine findet sich auf $FC53.

Da bei READ der „Denibble"-Prozeß mit inbegriffen ist, werden die folgenden 256 Datenbytes in drei Teilen behandelt, die im Prinzip alle gleich ablaufen.

Das nächste Byte wird von der Disk gelesen, via EXOR RDTRANS übersetzt. Wir haben jetzt ein Byte der Form A7 A6 A5 A4 A3 A2 00 00 und müssen die fehlenden beiden Bits noch aus dem Puffer mit den „zusätzlichen" Bytes holen. Diese „zusätzlichen" Bytes haben den selben Index wie das momentan gelesene Byte, d.h. zu Byte 00 der Daten gehört ein Teil des Byte 00 im „zusätzlichen" Puffer, zu Byte 01 der Daten gehört ein Teil von Byte 01 etc.
Um diesen Teil jetzt „herauszufiltern" und an das Byte anzuhängen, exisitiert eine schlichte (und deshalb umfangreiche) 1:1-Tabelle, die aus drei ineinander verschachtelten Tabellen besteht.

Ein Beispiel:
Das „zusätzliche" Byte habe den Wert 00 01 10 00, wir befinden uns im ersten Drittel des Datenpuffers und benötigen deshalb die Bits 3 und 2, um ein vollständiges Datenbyte zu erhalten.
Der Gesamtwert des „zusätzlichen" Byte wird als Index innerhalb einer Tabelle mit der Basisadresse $FA00 genommen. Diese Tabelle hat folgenden Inhalt:

```
FA00- 00 ** ** ** 02 ** ** **  01 ** ** ** 03 ** ** **
FA10- 00 ** ** ** 02 ** ** **  01 ** ** ** 03 ** ** **
FA20- ....
```

Egal, welchen Inhalt das „zusätzliche" Byte hat, durch die Bits 3 und 2 werden immer nur die Werte $00/$01/$02/$03 adressiert. Da beim „Prenibbling" die Reihenfolge dieser Bits umgetauscht wurde, um kompatibel mit DOS 3.3 (und dem Controller-Bootprogramm) zu bleiben, steht die Reihenfolge hier wiederum „verkehrtherum". In unserem Beispiel erhalten wir den Index $18 und damit den Wert $01.

Für das zweite Drittel des Datenpuffers hat die Tabelle die Basisadresse $FA01:

```
FA00- ** 00 ** ** ** 00 ** **  ** 00 ** ** ** 00 ** **
FA10- ** 02 ** ** ** 02 ** **  ** 02 ** ** ** 02 ** **
FA20- ....
```

Hier werden durch die Bits 4 und 5 jeweils die Werte $00/$01/$02/$03 adressiert. Da je vier der durch Bit 3/2 adressierbaren Werte gleich sind, spielt hierfür der Wert dieser beiden Bits keine Rolle.

Für das dritte Drittel müßte man eine halbe Seite volltippen: hier sind jeweils 16 Werte (in Vierer-Abständen) innerhalb der Tabelle gleich, so daß Bit 2/3 und Bit 4/5 ignoriert werden. Diese Tabelle hat die Basis $FA02.

Nur der Vollständigkeit halber sei an dieser Stelle noch einmal erwähnt, daß die WRTRANS-Tabelle, die ja auch in Vierer-Abständen aufgebaut ist, die Basis $FA03 hat und damit die restlichen 64 Byte auf dieser Speicherseite belegt.

# 5.4. HARDWARE DETAILS – Lesen und Schreiben von Bytes und die Steuerung des Drives

Anmerkung: Bei den folgenden Ausführungen ist das (teilweise) Abschreiben aus „Beneath Apple DOS" von Worth & Lechner, Kapitel 6 „Using DOS from Assembly Language" praktisch unvermeidlich. Apple, Inc. hat bis jetzt keine Informationen über die I/O-Speicherstellen auf dem Disk-Controller veröffentlicht. Auf eine entsprechende Anfrage des Autors im Jahre 1980 reagierte die „Hot line" in München etwa so: „Die Informationen haben wir nicht. Wenn Sie's wissen wollen: Beneath Apple DOS. Gibt's bei Pandasoft in Berlin. Steht alles drin. Guten Tag."

Die Standard-Controllerkarte läßt sich in zwei voneinander völlig getrennte Teile gliedern:

1) Das Boot-ROM. Enthält das Programm, das (unter Benutzung des zweiten Teils) bei einem PR#x oder Kaltstart des APPLE den Sektor 0 von Track 0 in den Speicherbereich $0800-$08FF liest. Dieses Programm ist im kommentierten Listing enthalten und in der „Blockbeschreibung" ausführlich besprochen.

2) 16 „Speicherstellen" (I/O-Adressen), über die der Datenverkehr und die Steuerung des Drives stattfinden. Diese I/O-Adressen sind abhängig von der Nummer des Slots, in dem sich die Controller-Karte befindet.
Jedem Slot sind $10 Adressen zugeordnet, die Basis ist die Adresse $C080:

$C080-8F: Slot 0 : nur für die Language Card, hier hat das Ansprechen dieser Speicherstellen eine spezielle Wirkung (Bank-switching, ROMselect). Diese Speicherstellen sind in der Dokumentation von Apple, Inc. beschrieben.

$C090-9F: Slot 1
$C0A0-AF: Slot 2
....
$C0F0-FF: Slot 7

Das Ansprechen dieser I/O-Adressen geschieht bei „slotunabhängig" geschriebenem Code über die berühmte „Slotno * 16", d.h. wenn zum Beispiel die zweite I/O-Adresse einer Karte angesprochen werden soll ($C081 für Slot 0):

```
        LDX  #(Slotno * 16)
        LDA  $C081,X
```

Wie man sieht, ergibt sich die spezifische Kartenadresse mit X=$00 für Slot 0, X=$10 für Slot 1, X=$20 für Slot 2 etc.

Die 16 I/O-Adressen auf dem Controller werden hier nur tabellarisch zusammengefaßt und dann einzeln erläutert, die aufgeführten Adressen stellen die Basisadresse dar, d.h. sie sind „Slot-los":

$C080-$C087 : Phasen des Stepper-Motors OFF ($80/$82/$84/$86) und ON ($81/$83/$85/$87).
$C088 : MOTOR ON
$C089 : MOTOR OFF
$C08A : Select Drive 1
$C08B : Select Drive 2
$C08C : Strobe/Input (Abschicken eines Befehls/Lesen eines Datenbytes)
$C08D : Laden des Zwischenspeichers (Byte => Disk)
$C08E : READ (Setzen der Richtung des Datentransfers)
$C08F : WRITE (Setzen der Richtung des Datentransfers)

Die meisten dieser Adressen müssen lediglich mit irgendeiner Instruktion (LDA/LDX/LDY, CMP oder BIT tut es auch) angesprochen werden, um den gewünschten Effekt auszulösen. Lediglich bei den Speicherstellen $C08C und $C08D kommt es auf den Inhalt an.

Bei der folgenden Besprechung wurde versucht, die Speicherstellen in der Reihenfolge abzuhandeln, wie sie in der RWTB benutzt werden. Bei den Beispielen wird wiederum davon ausgegangen, daß sich der Controller in Slot 6 befindet.

**A) Select Drive 1/2 und MOTOR ON/OFF**

Select Drive ist die einzige Möglichkeit, zwischen Drive 1 und Drive 2 zu unterscheiden. Alle anderen Befehle beziehen sich grundsätzlich auf das über diese beiden Speicherstellen selektierte Laufwerk.

```
Drive 1:    LDX #$60              ; Slotno * 16 für Slot 6
            LDA $C08A,X
```

```
Drive 2:    LDX #$60
            LDA $C08B,X
```

Danach folgt dann MOTOR ON:

```
            LDX #$60
            LDA $C089,X          ; MOTOR ON für den selektierten Drive
```

In der DISKII-RWTB ist das ganze etwas kompakter zusammengefaßt:

```
F854-  A5 43     LDA  $43        ; Unitnummer (DSSS 0000)
       ....                      ; X wird mit Slotno * 16 geladen
F85D-  0A        ASL             ; „D" aus der Unitnummer ins Carry
F85E-  BD 89 C0  LDA  $C089,X    ; MOTOR ON
F861-  90 01     BCC  $F864      ; kein Carry: Drive 1
F863-  E8        INX
F864-  BD 8A C0  LDA  $C08A,X    ; „$C08B,X" für Drive 2
```

Um jetzt noch die Funktion der verschiedenen Delays zu erklären, die vorher und nachher aufgerufen werden, müssen wir etwas in die Elektronik einsteigen.
Ein Motor ist ein mechanischen Gerät und unterliegt deshalb den Gesetzen der Masseträgheit, d.h. er braucht eine gewisse Zeit, bis er seine volle Drehzahl erreicht.
1) Da die RWTB (genauso wie entsprechende Routinen anderer Betriebssysteme) nicht „weiß", ob nach diesem Sektor/Block noch weitere Diskettenzugriffe erfolgen, wird der Motor nach dem Lesen/Schreiben des Sektors abgeschaltet, damit er nicht ewig läuft (bei einigen Steinzeit-Systemen ist das tatsächlich der Fall: die Drives sind an, solange der Computer eingeschaltet ist).
Würde dem Motor bei diesem „Abschalten" einfach die Stromversorgung unterbrochen, hätte das zur Folge, daß vor dem Lesen jedes weiteren Sektors gewartet werden muß, bis der Motor wieder auf Touren kommt, d.h. ein Diskettenzugriff über mehrere Sektoren hinweg würde eine halbe Ewigkeit dauern.
Um das zu verhindern, sitzt zwischen dem softwaremäßig bedienbaren „Abschalter" und der tatsächlichen Abschaltung des Motors ein elektronisches Verzögerungsglied, ein sogenanntes Monoflop. Jedesmal, wenn der Motor vom Programm her abgeschaltet wird, läuft er noch eine Weile (ca. 1.8 Sekunden).
Erfolgt innerhalb dieser Zeit ein weiterer Zugriff, wird der Motor softwaremäßig wieder angeschaltet und ist dann sofort (immer noch) auf voller Drehzahl. Ist dieser Zugriff dagegen der erste, d.h. war der Motor des Drives tatsächlich aus, wird das „Motor-on Delay" ausgeführt.
2) Das Netzteil des APPLE hat seine Grenzen: Es ist ihm nicht sonderlich zuträglich, wenn gleichzeitig mehr als ein Diskettenmotor läuft. Wird z.B. zuerst auf eine Disk in Slot 5 zugegriffen und direkt danach auf eine Disk in Slot 6, dann läuft der Motor des Drives in Slot 5 noch aufgrund der erwähnten Abschaltverzögerung. Hier muß gewartet werden, bis der Motor im „alten" Slot wirklich aus ist. Wird dagegen innerhalb desselben Slots von einem Drive auf den anderen umgeschaltet, findet eine echte Umschaltung der Stromversorgung statt, diese Verzögerung ist dann nicht nötig.
3) Egal, ob von Drive 1 auf Drive 2 innerhalb desselben Controllers oder zwischen zwei verschiedenen Controllern umgeschaltet wird: Der Schreib/Lesekopf muß auf die Diskettenoberfläche abgesenkt werden (Head Load). Auch das braucht seine Zeit. (Anmerkung: Bei einigen Laufwerken liegt der Kopf nach dem Schließen der Klappe ständig auf der Diskette

(DISK II, FD-100 usw.), bei anderen Typen senkt ein Kopflademagnet den Schreib-/Lesekopf ab, wenn das Laufwerk angesprochen wird.)

Ein Flußdiagramm des Ablaufs innerhalb der RWTB dazu:

| | |
|---|---|
| $F848- | Aufruf von SAMESLOT? ($FE9B) |
| $FE9F- | Ist das jetzt gewünschte Unit im selben Slot wie das zuletzt benutzte, RTS. |
| $FEAC- | Ist überhaupt kein „zuletzt benutztes Unit" verzeichnet, RTS. |
| $FEAE- | Aufruf von MOTOR ON? ($FCDC) für das „alte" Unit. |
| $FCDC- | Testet, ob sich die Daten am Lesekopf des Units ändern. Wenn ja, ist der Motor tatsächlich (noch) an. |
| $FEB1- | Hat der Test ergeben, daß der Motor aus ist, RTS. Ansonsten wird der Test so oft wiederholt, bis der Motor aus ist (Verzögerungszeit abgelaufen.) |
| $F84F- | Die Zeitkonstante für das Motor-on Delay wird immer gesetzt. |
| $F85E- | Der Motor des gewünschten Drives wird angeschaltet, der Drive selektiert. |
| $F867- | Wenn „altes" und „neues" Unit nicht identisch sind: Verzögerung, um dem Schreib/Lesekopf des neuen Laufwerks Zeit zum Absenken zu geben (Head Load Time). |
| $F881- | Wenn „altes" und „neues" Unit nicht identisch sind oder der Motor wirklich abgeschaltet war: Ausführung des Motor-on Delay. Danach Test, ob der Motor jetzt seine Nenndrehzahl erreicht hat ($FCDA). |

– Ausführung des Befehls

| | |
|---|---|
| $F8F0- | „Abschalten" des Motors und RTS. |

**B) Bewegung des Arms (SEEKABS und „Arm Mover")**

Der Arm des Drives wird mit einem Stepper-Motor (Schrittmotor) bewegt. Bei moderneren Laufwerken erledigt das die interne Elektonik des Drives selber, hier werden nur noch Impulse (Step in / Step out) gesendet. Bei der DiskII erfolgt die Ansteuerung des Stepper-Motors von der RWTB aus. Darum kann man damit auch einen Haufen Unsinn anfangen („Half"- und „Quarter"-Track „Kopierschutz").
Für den Stepper-Motor sind acht I/O-Adressen zuständig, die „geraden" Adressen ($C080/82/84/86) stehen für PHASE OFF, die „ungeraden" ($C081/83/85/87) für PHASE ON.
Um den Arm über einen Track zu bewegen, müssen zwei Phasen hintereinander angeschaltet werden, die Reihenfolge bestimmt dabei die Bewegungsrichtung.

```
Step in:    LDX  #$60          ; Beispiel wieder mit Slot 6
            LDA  $C081,X       ; PHASE 0 ON
            LDA  $C083,X       ; PHASE 1 ON

Step out:   LDA  $C083,X       ; PHASE 1 ON
            LDA  $C081,X       ; PHASE 0 ON
```

Der Stepper-Motor besitzt insgesamt vier solcher Phasen, d.h. damit wäre eine Bewegung über zwei Tracks möglich. Um ihn über mehrere Tracks hinwegzubewegen, müssen die angeschalteten Phasen auch wieder ausgeschaltet werden, und zwar ebenfalls in definierter Reihenfolge. Um dabei eine definierte Bewegungsrichtung zu erhalten, wird die soeben angeschaltete Phase erst beim nächsten Schritt wieder abgeschaltet:

```
Step in:    LDA $C081,X          ; PHASE 0 ON
            LDA $C083,X          ; PHASE 1 ON
            LDA $C080,X          ; PHASE 0 OFF
            LDA $C085,X          ; PHASE 2 ON
            LDA $C082,X          ; PHASE 1 OFF
            LDA $C087,X          ; PHASE 3 ON
```

Ein Beispiel für die Bewegung des Arms von Track 0 nach Track 3, die Phasen in der Reihenfolge, wie sie dann tatsächlich in der RWTB angesprochen werden:

```
0 OFF / 1 OFF / 2 OFF / 3 OFF      ; $F925, alle Phasen OFF
1 ON  / 0 OFF / 2 ON  / 1 OFF      ; 0 OFF ist hier ohne Bedeutung
3 ON  / 2 OFF / 0 ON  / 3 OFF
1 ON  / 0 OFF / 2 ON  / 1 OFF
2 OFF                              ; der Ziel-Track ist erreicht
0 OFF / 1 OFF / 2 OFF / 3 OFF      ; $F925, alle Phasen OFF
```

Um die fortlaufenden Adressen zu erhalten, wird praktischerweise der „Track jetzt" genommen und während SEEKABS jeweils um eins auf den „Track gewünscht" hin erhöht/erniedrigt. Die Reihenfolge der Phasen ergibt sich damit von selbst.
Da der Stepper-Motor natürlich auch eine gewisse Zeit braucht, um sich zu bewegen, wird nach jedem Schalten einer Phase ein Delay angehängt. In dieser Form findet sich die Routine im Urlader ($09BC-$09E0 bzw. $096D-$0985). Hier hat man sich sogar noch das vorherige und nachherige Abschalten aller Phasen gespart, d.h. es ist möglich, daß eine oder zwei Phasen des Stepper-Motors unter Strom stehen, bis zum ersten Mal (ONLINE) die RWTB-eigene Routine aufgerufen wird. (Daher auch wahrscheinlich die Notwendigkeit, alle Phasen nicht nur nach, sondern auch vor einem Aufruf von SEEKABS erst einmal abzuschalten).

Innerhalb der RWTB-eigenen SEEKABS geht es etwas komfortabler (und wesentlich komplizierter) zu: Wenn der Stepper-Motor einmal in Bewegung ist, können die Wartezeiten zwischen den einzelnen Phasen-Schaltungen verkürzt werden. Je mehr Bewegungen der Motor bereits hinter sich hat, desto schneller ist er. Allerdings muß er, wenn der Arm sich dem Ziel-Track nähert, auch wieder abgebremst werden. Zu dieser Geschwindigkeitssteuerung gehören zwei Dinge: eine Tabelle mit den Delay-Werten und ein Index darin. Dieser Index wird vor der ersten Bewegung auf Null gesetzt und zeigt damit auf die maximale Verzögerungszeit. Für jede Bewegung wird der Index danach erhöht, bis er entweder den Wert 8 überschreitet (Tabellenende) oder bis der Abstand zum Ziel-Track kleiner geworden ist als der momentane Index. Ab da dient der (pro Bewegung kleiner werdende) Abstand selber als Index, d.h. die Verzögerungszeiten werden wieder größer.
Wenn schließlich der Ziel-Track erreicht worden ist, wird die Delay-Routine einmal mit dem Maximalwert (00) aufgerufen, um dem Arm Zeit zu geben, sich „auszupendeln" (Head Settling Time).
Da für den Index der Delay-Werte zuerst auch noch der Absolutwert der (neuen!) Differenz zwischen „Track jetzt" und „Track gewünscht" berechnet werden muß, ist die SEEKABS-Routine in der RWTB wirklich ein gründliches Studium wert. (Sie ist allerdings nicht vom

ProDOS-Programmierer geschrieben, sondern fast unverändert von DOS 3.3 übernommen worden, zudem ist in der ProDOS-Version noch ein Fehler drin, der aber nicht auffällt – siehe „Spezielles").
Eine Kleinigkeit ist noch unterschlagen worden: Die Delay-Werte für PHASE ON und PHASE OFF sind verschieden, deshalb gibt es zwei Tabellen für diese Werte.

### C) Lesen einens Bytes von der Diskette

Zuerst muß READ-Mode gesetzt werden. Dieser Modus ist normalerweise immer gesetzt.

```
Set READ:    LDX  #$60
             LDA  $C08E,X          ; Setzt INPUT
```

Wie durch Geisterhand erscheinen von nun an Datenbytes in der I/O-Adresse $C08C(,X).
Zur Erklärung ist wieder einmal ein bißchen Elektronik nötig:
Auf der Diskette sind die Bytes bitweise hintereinander gespeichert. Die Art und Weise, wie das geschieht (und die damit verbundenen Probleme der Trennung von Daten- und Clockbits, Phase locked loops etc.) brauchen uns hier Gott sei dank nicht zu interessieren. Wir betrachten einfach einen Bit-Strom, der am Lesekopf des Drives vorbeiläuft. Jedes Bit (0 oder 1), das vorbeikommt, wird in ein Schieberegister hineingeschoben. Ist das Schieberegister nach dem Hineinschieben des achten Bits voll, wird es gelöscht, bevor ein neuntes Datenbit heineingeschoben wird, d.h. der Prozeß fängt von vorne an. Eine Darstellung dieses Prozesses für Nicht-Elektroniker, die vom logischen Ablauf her fast identisch mit diesem elektronischen Prozeß ist:

```
START:       LDX  #$08             ; erster Aufruf: Zähler für 8 Bit
             LDA  #$00
             STA  SHIFTREG         ; Schieberegister löschen

NEXT-BIT:    JSR  GET-BIT
             ROL  SHIFTREG         ; Carry in das Schieberegister
             DEX
             BNE  NEXT-BIT
                                   ; **
             LDX  #$07             ; ** In dieser Zeit muß
             JSR  GET-BIT          ; ** der Inhalt von SHIFTREG
             LDA  #$00             ; ** „abgeholt" werden!
             STA  SHIFTREG
             ROL  SHIFTREG
             JMP  NEXT-BIT

GET-BIT:     SEC wenn gefundenes Datenbit = 1, sonst CLC
             RTS
```

Diese Darstellung hat allerdings mehrere Haken: zum einen ist der 6502 schlicht zu langsam, um diesen Prozeß auszuführen (der C 64 zum Beispiel macht so etwas ähnliches mit seiner seriellen Kommunikation zur Diskette – seine Besitzer wissen ein trauriges Lied davon zu singen). Auf dem APPLE „dauert" ein Datenbit 4 Mikrosekunden, d.h. alle 32 Mikrosekunden ist das Schieberegister voll.
Der zweite Haken: die Elektronik hat keinen Zähler, der 8 Datenbits herunterzählt. Die Bestimmung „Schieberegister voll" erfolgt wesentlich einfacher – es wird jedesmal gelöscht, wenn das Bit 7 „links herausgeschoben" wird. Noch ein Programm-Äquivalent dazu:

```
NEXT-BIT:    JSR  GET-BIT
             ROL  SHIFTREG          ; hineinschieben
             BCC  NEXT-BIT          ; SHIFTREG war noch nicht voll
NEXT-BYTE:   ROR  SHIFTREG          ; SHIFTREG ist voll, das neue Bit zurück!
             LDA  #$00
             STA  SHIFTREG
             ROR  SHIFTREG          ; und das neue Bit hinein
             JMP  NEXT-BIT
```

Eine noch nicht erklärte Voraussetzung ist damit eingeführt worden: das Bit 7 **jedes** Bytes muß gesetzt sein. Auf andere Weise ist es unmöglich zu erkennen, wann das Schieberegister voll ist. Sogenannte „zeitkritische" Codes auf der Diskette (ein „Kopierschutz", der lange Zeit weit verbreitet war) haben das Bit 7 nicht gesetzt. Hier wird einfach 32 Mikrosekunden gewartet und dann „zugeschlagen". Die entsprechenden Probleme bei Laufwerken mit unterschiedlichen Motorgeschwindigkeiten („Boot-Probleme") sind bekannt.
Machen wir das Schieberegister auf dem Controller wieder zur „black box": Die Elektronik sorgt dafür, daß der Prozessor beim Abfragen der I/O-Adresse $C08C(,X) den momentanen Inhalt des Schieberegisters zu lesen bekommt.

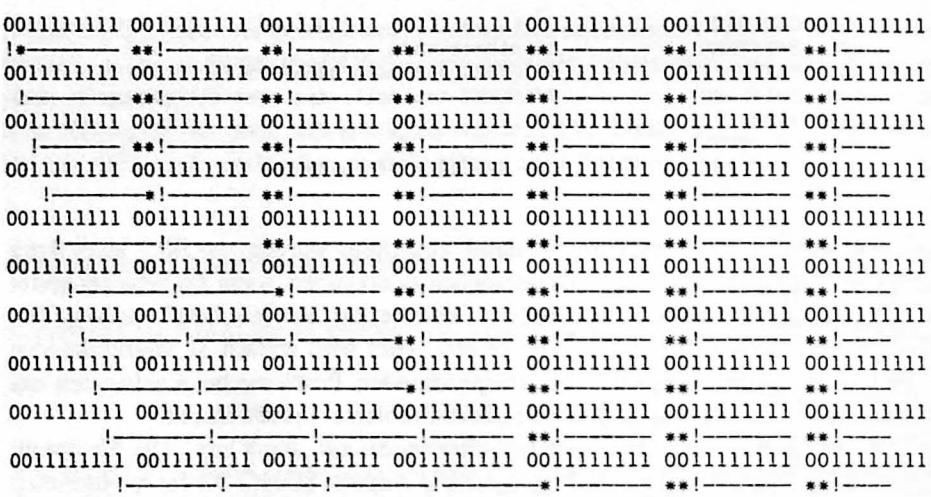
Das Lesen eines Bytes vom Programm her sieht dann folgendermaßen aus:

```
SET-READ:    LDX  #$60              ; Slot 6
             LDA  $C08E,X           ; READ Mode
             ....
GET-BYTE:    LDA  $C08C,X
             BPL  GET-BYTE          ; solange, bis Bit 7 gesetzt ist
GOT-BYTE:    ....
```

Das funktioniert hervorragend, sobald der Controller „in Sync" ist. Auf der Diskette findet sich in etwa folgender Datenstrom:

01101101110110110110110101110111011010101010110110110110101011011111011

Der Drive wird selektiert, der Motor angeschaltet und der Kopf abgesenkt. Die Elektronik (und damit der Prozessor) fängt jetzt mitten in diesem Datenstrom an, Bits in Achter-Blökken zu sammeln. Die Wahrscheinlichkeit, daß hier genau mit dem Anfang eines „richtigen" Bytes begonnen wurde, ist ungefähr 1:8.
Um den Anfang eines Bytes zu bestimmen, hat man sich eine spezielle Sorte von Bytes auf der Diskette ausgedacht, die sogenannten „Autosyncs". Diese Bytes bestehen aus acht „Einsern" ($FF) und zwei folgenden „Nullen". Da die Elektronik „führende" Nuller ignoriert, weil sie ein gesetztes Bit 7 im Datenbyte erwartet, bevor das Schieberegister gelöscht wird, kann man mit diesen Bytes erreichen, daß sich die Elektronik praktisch von selber „einhängt".
Die folgende Tabelle ist praktisch unverändert von Worth & Lechner übernommen, ein Sternchen steht für vom Controller ignorierte Bits, ein Ausrufezeichen markiert den Start der nächsten Leseaktion:



Woraus folgt: stehen mehr als vier dieser Autosyncs hintereinander auf der Diskette, hat sich die Elektronik nach dem Lesen mit Sicherheit „eingeklinkt" (Neudeutsch: ist in Sync). Deshalb stehen vor jedem Address Field und vor jedem Data Field vor dem eigentlichen Header (D5 AA 96/AD) fünf Autosyncs. Das Address Field wird bei der Formatierung geschrieben und von da an nur noch gelesen, beim Data Field werden diese 5 Autosyncs bei jedem Überschreiben des Inhalts ebenfalls neu geschrieben. Die restlichen Daten schließen sich zeitlich lückenlos an.
Noch eine wichtige Anmerkung: Die Elektronik auf dem Controller arbeitet selbständig, d.h. der Prozessor ist zwar dafür verantwortlich, daß die Daten rechtzeitig „abgeholt" werden, aber die Elektronik bleibt „in Sync", solange die Daten auf der Diskette kontinuierlich geschrieben sind (also normalerweise bis zum Ende des momentanen Address oder Data Field).

### D) Schreiben eines Bytes und WRITE PROTECTED

Sieht man sich die I/O-Adressen der Controller-Karte der Reihe nach an, wird man eine spezifische Adresse vermissen, durch deren Lesen das Programm testen könnte, ob die Zieldiskette schreibgeschützt ist. Da innerhalb der $10 zur Verfügung stehenden Adressen auch keine mehr übrig ist, wird hier etwas von der Logik abgewichen und per definitionem eine „Fehlschaltung" erzeugt:

```
SENSE-WPROT: LDX  #$60            ; Slot 6
             LDA  $C08D,X         ; das ist eigentlich der OUTPUT
             LDA  $C08E,X         ; set READ
```

Ist die Zieldiskette WRITE PROTECTED, ist das Bit 7 von A jetzt gesetzt:

```
             BMI IS-PROTECTED
```

Das Schreiben von Daten auf die Diskette:

```
SET-WRITE:   LDX  #$60          ; Slot 6
             LDA  $C08F,X       ; set WRITE
             .....

WRITE-BYTE:  STA  $C08D,X       ; Daten zur Disk
             LDA  $C08C,X       ; Strobe
```

Das Schreiben funktioniert quasi umgekehrt wie der Leseprozeß: Das Schieberegister wird in diesem Fall beladen (allerdings ist hier aus „elektronischen Gründen" die Adresse eine andere) und danach durch einen „Strobe" abgeschickt.
Die Elektronik fängt in dem Moment an, das Datenbyte sequentiell auf die Diskette zu schreiben, wo die Speicherstelle $C08C(,X) adressiert wird. Die Adresse $C08D(,X) ist danach aus programmtechnischer Sicht „leer", sie kann auch nicht vom Prozessor gelesen werden.
Der Haken an der Sache: Im Gegensatz zu READ, wo der Prozessor schlicht darauf wartet, daß das nächste Datenbyte erscheint, ist bei WRITE der Programmierer dafür verantwortlich, daß alle 32 Mikrosekunden ein neues Datenbyte bereit steht, d.h. daß der Strobe alle 32 Mikrosekunden einmal ausgelöst wird. (Ausnahme: Autosyncs, hier ist der zeitliche Abstand 40 Mikrosekunden. Das Schieberegister „läuft einfach leer" und es werden damit automatisch zwei Nuller an das geschriebene Datenbyte angehängt).
WRITE hat eine weitere Eigenschaft, die sich bei „wildgewordenen" Programmen, die auf dieser Ebene arbeiten, katastrophal auswirken kann: Wird das Schieberegister nicht mit neuen Daten beladen, schreibt die Elektronik einfach „Nuller" – unter Umständen einen ganzen Track. Da damit auch die Address Fields zerstört werden, muß zumindest der Track komplett neu formatiert werden.
Generell gilt: Die Bits müssen in Abständen von 4 Mikrosekunden geschrieben werden, ansonsten ist der Sektor nicht mehr lesbar.

Die WRITE SECTOR-Routine in der RWTB lokalisiert einen Sektor via Lesen des Address Field, schreibt dann fünf Autosyncs, gefolgt vom Data Field Header. Danach kommen die Daten selber und der Trailer. Sofort danach muß wieder auf READ umgeschaltet werden, ansonsten überlebt das Address Field des folgenden Sektors diese Aktion nicht. Während des gesamten Schreibvorgangs muß das Timing auf die Mikrosekunde genau konstant gehalten werden.

Tucholsky hat einmal gesagt: „Schreiben ist zu jeder Zeit gefährlicher gewesen als Lesen". Es ist allerdings verbrieft, daß er keinen APPLE auf seinem Schreibtisch hatte.

## 5.5 Blockbeschreibung des Drivers (RWTB) für die DISKII

Der Aufruf geschieht vom MLI aus grundsätzlich (auch für BLOCKREAD/WRITE) über $D0DA. Hier wird via Unitnummer die Startadresse des Drivers aus der Global Page geholt (wobei interessanterweise die Startadressen für D1/D2 desselben Slots verschieden sein können), in $F0B5/B6 gespeichert und via JMP ($F0B5) zur RWTB gesprungen.

Dabei sind folgende Speicherstellen gesetzt:

| | |
|---|---|
| $0042 | – Command: 0 – Select; 1 – READ; 2 – WRITE. |
| $0043 | – Unitnummer (DSSS 0000). |
| $0044/45 | – Ziel/Quelladresse im Speicher (lo/hi). |
| $0046/47 | – Blocknummer (lo/hi). |

Bei BLOCKREAD/WRITE nimmt das MLI einen Platzbedarf von $200 an ($300, wenn die Ziel/Quelladresse nicht auf einer Seitengrenze anfängt) und prüft diesen Speicherbereich gegen die System Bit Map (auch für WRITE!).
Bei WRITE als MLI-Command wird dagegen der Speicherbereich nicht geprüft, man kann also mit OPEN und WRITE auch das MLI selber(!) abspeichern.

Die Ziel/Quelladresse ist bei allen MLI-internen Aufrufen immer auf einer Seitengrenze ($xx00). Nur für BLOCKREAD/WRITE ist der niederwertige Teil der Ziel/Quelladresse beliebig – um diese Möglichkeit offen zu halten, wurde ein fast unverhältnismäßig hoher Aufwand getrieben.

Die Kompliziertheit der Nibble/Denibble-Routinen hat einen ganz bestimmten Grund:
Zum einen wurde versucht, die Lesegeschwindigkeit gegenüber der RWTS von DOS 3.3 zu erhöhen. Das ist hier (auf Sektor-Ebene) im Prinzip auch gelungen. Unter DOS wurde ein Sektor erst in einen RWTS-eigenen Puffer ($BB00-$BC55) gelesen und danach während des Denibbling in den Zielbereich kopiert.
Die ProDOS-RWTB liest dagegen direkt in den Zielbereich, nur für die „zusätzlichen" $56 Nibbles wird ein eigener Puffer ($FB00-$FB55) zur Verfügung gestellt. Das Denibbling geschieht **während** des Lesens.
Zum anderen: Dieser Prozeß wäre mit einem neuen Disk-Format um einiges einfacher gewesen. Erfreulicherweise hat Apple, Inc. aber für ProDOS (bis jetzt) darauf verzichtet, dem Benutzer eine „zwei-Formate-Disk" zu präsentieren. (Ohne ein zweites Format wäre es nicht gegangen: das Boot-Programm des Disk-Controllers kann nur DOS-Formate lesen, man hätte also eine CP/M-Analogie schaffen müssen: ein „Standard"(DOS)-Format für die System-Tracks, ein neues Format für den Rest der Diskette. Für APPLE-Freaks heißt so etwas „kopiergeschützt").

Bei der folgenden Beschreibung wurde wiederum von einem APPLE mit 64(128k) Speicher und einem Controller in Slot 6 ausgegangen.
Die RWTB liegt im File PRODOS bereits in der auf $F800-$FEFF relokalisierten Form vor, für einen APPLE mit 48k RAM werden diese Routinen auf $B800-$BEFF relokalisiert.

$F800 – Einziger definierter Einsprung in die RWTB. A,X,Y und P-Register des 6502 sind undefiniert. Beim Aufruf über BLOCKREAD/WRITE sind die Interrupt bereits abgeschaltet, bei MLI-internen Aufrufen sind sie es (noch) nicht!
Die Blocknummer wird aus $0046/47 geladen und gegen die Obergrenze für die DiskII geprüft. Überschreitet sie $0117 (Start der Zählung: Block 0000), Sprung zu $F836.

Eine Anmerkung: Hier offenbart sich wieder einmal die verquere Denkweise des ProDOS-Programmierers. Ein direkter Vergleich (CPX #hi+1 / BCS ERR / BNE OK / CMP #lo+1 / BCS ERR) hat eine veränderbare Obergrenze und ist glatt ein Byte länger.

$F810 Durch 5maliges Links-Shiften der Blocknummer ergibt sich die Tracknummer in $FB56 (5 * links entspricht hier 3 mal rechts, d.h. Teilung durch 8).
In A steht in den Bits 5-7 der Divisionrest (Bereich: 0-7). Durch einen weiteren Shift wird bestimmt, ob der Rest im Bereich 0-3 oder 4-7 liegt. Ist der Divisionsrest im Bereich 4-7, wird Bit 4 gesetzt.
Es folgen 4 Shifts nach rechts, mit folgendem Ergebnis:

Divisionsrest (Blocknummer innerhalb eines Tracks):

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 0(2) | 4(6) | 8(A) | C(E) | 1(3) | 5(7) | 9(B) | D(F) |

Erster Sektor des Blocks (physikalisch). In Klammern: der zweite Sektor des Blocks.

$F822 Die so gewonnene Sektornummer wird zwischengespeichert, der Sektor wird gelesen ($F83A). Bei Fehler Sprung zu $F832. Die Sektornummer wird um zwei erhöht, die Quell/Zieladresse um $100. Zweiter Aufruf von $F83A, um den zweiten Teil des Blocks zu lesen. Die Ziel/Quelladresse wird danach wieder auf ihren ursprünglichen Wert zurückgesetzt.

$F832 Eine eventuell gesetzte ERR# wird geladen. RTS zur aufrufenden Routine (hier immer zur Routine, die $D0DA aufgerufen hat) mit gesetztem Carry und ERR# in A bei Fehler, ansonsten mit Carry gelöscht und dem Wert 00 in A.

$F836 ERR#27: I/O ERROR, wenn die maximal zulässige Blocknummer überschritten wurde. RTS zur Routine, die $D0DA aufgerufen hat.

$F83A **READ A SECTOR** – Aufruf mit Sektornummer in A.
Der 'recal count' (Zähler für die maximale Anzahl von Arm-Rekalibrierungen) wird auf 1 gesetzt, die Sektornummer gespeichert. Aus der Unitnummer auf $43 (DSSS 0000) ergibt sich nach Maskierung von Bit 7 die Slotnummer * 16. Sie wird auf $3E gespeichert.

$F848 Aufruf von $FE9B: die Slotnummer wird mit der Nummer des zuletzt benutzten Slots („OLDSLOT") verglichen; ist auf einem anderen Controller noch ein Drive an, wird gewartet, bis der Motor ausgegangen ist (Grund: Belastung des Netzteils).
Test, ob der gewünschte Drive (noch) an ist ($FCDA), das Ergebnis (NE, falls an) wird gespeichert.

$F84F Die Konstanten für das Motor-on-Delay werden immer gesetzt.

$F854 Das zuletzt benutzte Unit wird mit dem jetzt benutzten Unit verglichen und dabei gleichzeitig diese Speicherstelle (OLDSLOT/OLDDRIVE) auf den neuen Stand gebracht. Das Ergebnis (EQ, falls beide Units gleich sind) wird ebenfalls gespeichert.

$F85D Das Bit 7 der Unitnummer (Drive) wird ins Carry geschoben. Der Motor wird angeschaltet und über das Carry wird entweder Drive 1 ($C08A + Slot * 16) oder Drive 2 ($C08A + Slot * 16 + 1) selektiert.

$F867 Das Ergebnis vom Vergleich 'old unit - new unit' wird zurückgeladen. Waren beide Units gleich, Sprung zu $F874. Ansonsten wird auch das „Drive was on"-Flag vom Stack geholt, via $FB85 verzögert, um dem Schreib/Lesekopf Zeit zum Absenken zu geben („Head Load Time"). Danach wird das „Drive was on"-Flag wieder gespeichert. Es ist dann auf „Drive was off" (EQ) gesetzt.

$F874 **CORRECT DRIVE SELECTED**
Ist der Befehl ($42) = $00, Sprung zu $F87E. Für diesen Befehl findet keine Positionierung des Arms statt.
Ansonsten Aufruf von MYSEEK ($F90C), der Arm wird auf den gewünschten Track bewegt. Das dabei benötigte Delay zählt ebenfalls das „Motor-on-Delay" hoch, d.h. die Zeit, die für Armbewegungen gebraucht wird, wird in die Hochlaufzeit des Motors mit einbezogen.

$F87E Das „Motor was on"-Flag wird vom Stack geholt. War der Motor des gewünschten Drives bereits an (NE), Sprung zu $F890.
Sonst wird mit das „Motor-on-Delay" aufgeführt. Danach wird festgestellt, ob der Motor läuft, d.h. ob sich die Daten am Lesekopf ändern ($FCDA). Tut sich hier nichts, Sprung zu $F8EC mit von $FCDA gesetzem ERR#28: NO DEVICE CONNECTED.

$F890 **CORRECT DRIVE ON**
Für das $00-Command: Sprung zu $F8FD (Test auf WRITE PROTECTED und RTS).
Für WRITE wird die Prenibble-Routine ($FDF0) aufgerufen.

$F89A **TRY TRACK**
Der Retry-Counter (Fehlerzähler, bei 0 wird rekalibriert) wird auf $40 gesetzt.

$F89F RDADDR ($FB98) wird aufgerufen. Falls ein Address Field fehlerfrei gelesen wurde, Sprung zu $F8C0.

$F8A6 Der Retry-Counter wird heruntergezählt. Bleiben noch Retries übrig, Loop zu $F89F.

$F8AB Der Retry-Counter ist auf Null: Der recal count wird um eins erniedrigt. Wurde bereits rekalibriert (recal count jetzt = $FF), dann Exit mit ERR#27: I/O ERROR.
Ansonsten wird der „Track gewünscht" (* 2) kurzzeitig um 8 erhöht, um den Arm von diesem anscheinend schadhaften Track wegzubewegen. Der Retry-Counter wird wieder auf $40 gesetzt. Sprung zu $F8CE.

$F8C0 Sprung hierher, wenn ein Address Field fehlerfrei gelesen wurde. „Track gefunden" und „Track gewünscht" werden verglichen. Stimmen beide überein, Sprung zu F8D7.

$F8C8 **SETTRACK II**
Der „Track gewünscht" wird zwischengespeichert, „Track gefunden" wird geladen.

$F8CE Einsprung für die Rekalibrierung: „Track gefunden" ist auf „Track gewünscht" + 8 gesetzt.
Der „Track gefunden" wird als „Track jetzt" gesetzt ($FCD3), der „Track gewünscht" wieder zurückgeholt und MYSEEK ($F90C) aufgerufen, um den Arm auf den gewünschten Track zu bewegen. Bei einer Rekalibrierung des Arms wird der Arm dabei immer um 8 Tracks nach innen (zu höheren Tracknummern hin) bewegt, um ihn von einem anscheinend schadhaften Track wegzubekommen. (Beim nächsten Read eines Address Fields wird dann festgestellt, wo sich der Arm tatsächlich befindet und SETTRACK II noch einmal durchlaufen, dieses Mal mit den korrekten Tracknummern).

$F8D5 Nach dem Neusetzen der Arm-Position folgt Loop zu $F89F.

$F8D7 **ON CORRECT TRACK**
Die „Sektornummer gefunden" wird mit der „Sektornummer gewünscht" verglichen. Stimmen die beiden Werte nicht überein, Loop zu $F8A6. Das Command wird erneut geladen. Für WRITE folgt Sprung zu $F8F4.

$F8E4 Aufruf von READ DATAFIELD ($FBFD). Tritt dabei ein Fehler auf, Loop zu $F8A6.

$F8E9 A wird mit der ERR# 00 geladen (kein Fehler).

$F8EB „BNE $F925" – dieser Sprung wird nie ausgeführt.

$F8EC „SEC" im „Sprung" auf $F8EB versteckt. Wird als Error Exit von $F88E und WRITE benutzt.

$F8ED Die Fehlernummer wird gespeichert (00 = kein Fehler), der Motor wird abgeschaltet. RTS zu $F826/$F82D.

$F8F4 Aufruf von WRITE DATA FIELD ($FD00).

$F8F7 Wurde kein Fehler festgestellt, RTS via $F8E9.
Der einzige Fehler, der hier auftreten kann, ist WRITE PROTECTED. Da hier neue Versuche nichts nützen, wird in diesem Fall mit ERR#2B und Sprung nach $F8EC (unnötig, das Carry ist bereits gesetzt) sofort abgebrochen.

$F8FD-$F909 Test auf **WRITE PROTECTED** für das 00-Command
Die Slotnummer * 16 wird geladen, auf dem Zieldrive wird WRITE PROTECTED getestet. Das Ergebnis wird ins Carry rotiert (Bit 7 gesetzt: WRITE PROTECTED => Carry gesetzt). Sprung zu $F8F7.
Witzigerweise wird das $00-Command auch zweimal ausgeführt, weil die Unterscheidung erst innerhalb von READ/WRITE Sector stattfindet.

$F90C-$F932 **MYSEEK**
Steuer-Routine zur (absoluten) Armpositionierung. Aufruf mit „Track gewünscht" in A.

Anmerkung: Zur Armbewegung selber siehe „Hardware Details".
Die RWTB für die DiskII hat ein internes Verzeichnis, in dem die Armpositionen sämtlicher Units festgehalten werden, die über diese RWTB bedient werden. Diese Tabelle fängt auf $FB59 an.

$F90C Der „Track gewünscht" wird mal zwei genommen und gespeichert. Sämtliche Phasen des Stepper-Motors werden abgeschaltet ($F925). Aus dem Verzeichnis wird die alte Position des Arms via Unitnummer geholt und als „Track jetzt" gespeichert.

$F91C Wenn die Armbewegung beendet ist, sollte sich der Arm über dem jetzigen „Track gewünscht" befinden. Das Verzeichnis wird gleich hier auf den neuen Stand gebracht.

$F922 „Track jetzt" und „Track gewünscht" sind gesetzt. Aufruf von SEEKABS ($F933) zur tatsächlichen Armbewegung, danach weiter mit $F925.

$F925-$F932 Alle vier Phasen des Stepper-Motors werden durchlaufen und dabei abgeschaltet ($F98A), weil bei beiden Aufrufen das Carry gelöscht ist. Die während MYSEEK doppelt genommene Tracknummer ($FB5A) wird auf ihren logischen Wert (geteilt durch zwei) zurückgeshiftet. Beim ersten Aufruf von MYSEEK aus spielt das keine Rolle, die Speicherstelle wird erst hinterher beladen.
RTS entweder zu MYSEEK (erster Aufruf) oder zur Routine, die MYSEEK aufgerufen hat.

$F933-$F986 **SEEKABS**
Aufruf von MYSEEK mit gesetzem „Track jetzt" und „Track gewünscht". Beide Werte sind das Doppelte der logischen Tracknummer, jeder Loop innerhalb von SEEKABS führt zu einer Armbewegung über einen halben Track.

$F939 Sind „Track gewünscht" und „Track jetzt" bereits vor der ersten Armbewegung gleich, folgt Sprung zu $F987, wo eine Phase des Stepper-Motors angeschaltet wird. Das tut nur deshalb nicht weh, weil sie danach (ohne Delay) sofort wieder abgeschaltet wird ($F925), d.h. hier ist etwas daneben gegangen (s. „Spezielles").
Ansonsten wird der momentane Index in der Delay Table des Arm Mover auf 00 gesetzt.

$F940 Start des Loops: Ist die Differenz „Track jetzt minus Track gewünscht" negativ, wird „Track jetzt" um eins erhöht und das Vorzeichen der Differenz umgedreht; ansonsten wird „Track jetzt" um eins erniedrigt. Die Differenz zwischen den beiden Werten ist bereits der neuen Entfernung angepaßt (EOR #$FF => Umdrehen des Vorzeichens und -1; ADC #$FE mit gesetztem Carry entspricht ebenfalls einer Subtraktion von 1).

$F95A Die Differenz wird mit dem Index in der Delay Table des Arm Mover verglichen. Ist sie inzwischen wieder kleiner als der Index (d.h. der Arm nähert sich dem Ziel-Track) wird die Differenz selber als Index verwendet (die Delays werden größer). Ist die Differenz größer als der momentane Index, wird der Index erhöht (der Arm bewegt sich, die Delays

werden kürzer). Die Obergrenze für die Indizes ist 8, länger ist die Delay Table nicht.

$F967 Eine Phase in Richtung auf den „Track jetzt", der eben erhöht/erniedrigt wurde, wird angeschaltet ($F987, Carry gesetzt), danach folgt ein Delay mit dem gesetzten Delay-Index für PHASE ON ($FB85).

$F971 Vom letzten Loop ist noch eine zweite Phase an, sie wird jetzt ausgeschaltet ($F98A, Carry gelöscht); danach folgt ein Delay für PHASE OFF mit dem gesetzten Delay-Index ($FB85).

$F97E Der momentan mögliche Index innerhalb der Delay Table wird erhöht, Loop zu $F940.

$F983 Sprung hierher von $F94A: „Track gewünscht" und „Track jetzt" stimmen nunmehr überein. A enthält den Wert 00, d.h. die maximale Zeit für das nachfolgende Delay ($FB85). Das ist nötig, damit der Arm über dem Ziel-Track zur Ruhe kommt. Die letzte Phase, die in Richtung auf den Ziel-Track noch an ist, muß danach noch abgeschaltet werden, deshalb weiter mit $F987.

$F987-$F995 **ACTUAL ARM MOVER** – Adressierung der einzelnen Phasen des Stepper-Motors. Aufruf mit CLC: PHASE OFF, mit SEC: PHASE ON.

$F987 Einsprung zum Schalten einer Phase in Richtung auf den „Track gewünscht".

$F98A Einsprung zum Abschalten der Phase, die davor aktiv war (und von $F925: Abschalten aller Phasen).
Eine der vier Phasen wird mit Hilfe der laufenden Tracknummer adressiert. Je nach Status des Carry ergibt sich als Basis $C080/82/84/86 (OFF) oder $C081/83/85/87 (ON).
Nach dem Schalten wird das X-Register wieder mit der Slotno * 16 geladen. RTS zur aufrufenden Routine oder zu MYSEEK, wenn das das Ende von SEEKABS ist.

$F996-F9FF **READ TRANSLATION TABLE I** (RDTRANS1)
Innerhalb dieser Table sind drei Bit-Tables versteckt, die für die Prenible-Routine gebraucht werden: $F9A0-$F9A3/$F9C0-$F9C3 und $F9E0-$F9E3.
Außerdem liegt hier noch der Data Field Trailer (DE AA EB) „begraben", er befindet sich auf $F9C4-$F9C6.

$FA00-FAFF **READ TRANSLATION II UND WRITE TRANSLATION**
Die einzelnen Werte sind in Vierer-Abständen angeordnet, für READ ergeben sich die Basen $FA00 ($FA04/$FA08..), $FA01 ($FA05/$FA09..) und $FA02 ($FA06/$FA0A..).
Die WRITE TRANSLATION TABLE hat die Basis $FA03 ($FA07/$FA0B..) und enthält die gültigen Disk-Nibbles (Bereich: $96..$FF).

$FB00-$FB55 Puffer für die „zusätzlichen" $56 Nibbles pro Sektor.

| | |
|---|---|
| $FB56 | Die „Tracknummer gewünscht" (Blocknummer/8). Wird am Anfang der RWTB berechnet. |
| $FB57 | Die (physikalische) Sektornummer des ersten Sektors des gewünschten Blocks. |
| $FB58 | ERR# – auftretende Fehler werden hier gespeichert und vor dem Rücksprung ins MLI zurückgeladen. |
| $FB59 | Unitnummer des zuletzt benutzten Units. Wird jeweils auf den neuen Stand gebracht. |
| $FB5A | „Track jetzt" – der Track, über dem sich der Arm des Drives befindet. Wird bei SEEKABS zusammen mit der Armbewegung verändert. |
| $FB5B-$FB68 | Verzeichnis der letzten bekannten Armpositionen für sämtliche Units, die von dieser RWTB bedient werden. Die Werte sind die logischen Tracknummern * 2, die Reihenfolge innerhalb der Table ist S1/D1, S1/D2, S2/D1...S7/D1, S7/D2. |
| $FB69 | Retry Counter ($40) |
| $FB6A | Recal Counter ($01) |
| $FB6B | – SEEKABS: Anzahl der Armbewegungen, Delay-Index<br>– RDADDR: Re-seek Counter ($FC), d.h. max. 4 Re-seeks<br>– RDADDR: Scratch für Even-Odd decoding |
| $FB6C | Checksum in RDADDR |
| $FB6D..$70 | In RDADDR: Checksum/Sektor/Track/Volume |
| $FB6E | Sektornummer gefunden (RDADDR) |
| $FB6F | – RDADDR: Tracknummer gefunden<br>– MYSEEK: Scratch für „Track gewünscht"<br>– Sub-Zähler für Arm Mover Delay |
| $FB70 | – Zähler für Motor-on und Arm Mover Delay<br>– In SAMESLOT? ebenfalls als Zähler benutzt |
| $FB71 | Scratch in SEEKABS: zuletzt aktivierte Phase des Stepper-Motors |
| $FB72 | „Track gewünscht" * 2 (SEEKABS) |
| $FB73-$FB7B | PHASE ON Delay Table |
| $FB7C-$FB84 | PHASE OFF Delay Table |
| $FB85-$FB97 | Arm Mover Delay<br>Wird auch für Motor-on benutzt. Aufruf mit definiertem Zeitwert in A. Zählt bei Armbewegungen ebenfalls das Motor-on Delay ($FB70) hoch, d.h. wenn der Motor des Drives gerade erst eingeschaltet wurde, wird die Zeit für die Armbewegungen mit in die Hochlaufzeit des Motors eingerechnet. RTS zur aufrufenden Routine. |
| $FB98-FBFC | **RDADDR** – Lesen eines Address Field |
| $FB98 | Die Anzahl der Re-seeks wird auf 4 ($FC) gesetzt (Y-Register). |

| | |
|---|---|
| $FB9D | Das Y-Register wird um eins erhöht. Ist Y noch kleiner Null, Sprung zu $FBA5. |
| $FBA0 | Die Re-seeks werden um eins erhöht. Wird Null erreicht, RTS zur aufrufenden Routine (nur $F8A1) mit gesetztem Carry via $FBFB. |
| $FBA5 | Lesen eines Bytes von der Diskette. Hat das Byte nicht den Wert $D5, Sprung zu $FB9D. |
| $FBAF | Das erste Byte hatte den Wert $D5. Lesen des zweiten Bytes. Hat es nicht den Wert $AA, Sprung zu $FBAA (Test auf $D5). |
| $FBB8 | Das Y-Register wird mit dem Wert 3 geladen, bevor das dritte Byte gelesen wird. |
| $FBBA | Lesen des dritten Bytes. Hat es nicht den Wert $96, Sprung zu $FBAA (Test auf $D5). |
| $FBC3 | Die Interrupts werden (engültig) abgeschaltet. Eine Wiederherstellung des vorherigen INT-Status erfolgt erst in der Routine, die die RWTB (bzw. $D0DA) aufgerufen hat! |
| $FBC4 | Der Checksum-Zähler wird auf Null gesetzt. |
| $FBC9 | Acht Bytes werden gelesen, (even-odd) decodiert und in absteigender Reihenfolge als 4 Bytes gespeichert. |
| $FBE3 | Das zuletzt decodierte Byte (Checksum) muß mit dem mitlaufenden Checksum-Zähler den Wert 00 ergeben, ansonsten RTS mit gesetztem Carry. |
| $FBE6 | Lesen des Address Field Trailer. Nur die Bytes $DE/$AA werden überprüft. Bei Fehler RTS mit gesetztem Carry. |
| $FBF7 | RTS zur aufrufenden Routine mit Carry gelöscht. |
| $FBF9 | Error Exit: RTS mit Carry gesetzt. |
| $FBFD- | **READ DATAFIELD & DENIBBLE**<br>Aufruf mit Slotno * 16 in X. |
| $FBFD-$FC0E | Der folgende Code wird mit der Slotno * 16 modifiziert. |
| $FC0F-$FC30 | Im folgenden Code werden die drei Basisadressen für das erste/zweite/dritte Drittel innerhalb des Zielbereichs berechnet. Da die Leseroutine mit einem Index von $AA bzw. $AC (letztes Drittel) zu diesen Basen anfängt, verschieben sich die Basen um den entsprechenden Betrag nach unten. |
| $FC31 | $20 Retries werden gesetzt, der Datafield Header (D5 AA AD) wird gelesen. Werden dazu mehr als $20 Versuche gebraucht, RTS mit gesetztem Carry; der Motor bleibt dabei an. |
| $FC53 | Die „zusätzlichen" Bytes werden in den RWTB-eigenen Puffer ($FB00-$FB55) in absteigender Reihenfolge gelesen. Die „Checkum" ($003A) läuft dabei mit. |
| $FC69 | Der Index #$AA wird für das erste Drittel geladen, Sprung zu $FC72. |
| $FC6D | Error Exit: Die zulässige Anzahl der Retries wurde überschritten. |
| $FC6F-$FC82 | Loop zum Lesen des ersten Drittels |
| $FC83 | Das letzte Byte des ersten Drittels wird nur zwischengespeichert (Grund: Y = 00, das Byte gehört auf „Basis+$100"), der Index #$AA wird erneut geladen. |
| $FC88-$FC9B | Loop zum Lesen des zweiten Drittels |

$FC9C-$FCB8 Loop zum Lesen des letzten Drittels, der Startindex ist hier $AC.

$FCB9 Das letzte gelesene Byte ist hier die Checksum und wird nicht gespeichert. Ergibt dieser Wert mit der laufenden EXOR-Kette nicht $00, Sprung zu $FCCC.

$FCC0 Lesen des ersten Data Field Trailer Bytes ($DE). Die restlichen Bytes des Trailers werden nicht verifiziert. Stimmt der Wert, Sprung zu $FCCD.

$FCCC SEC – Fehleranzeige.

$FCCD Das letzte Byte des ersten Drittels, das nur zwischengespeichert wurde ($FC83), wird vom Stack geholt und dahin gespeichert, wo es hingehört: auf das $56.te Byte des Zielpuffers.

$FCD2 RTS zu $F8E7, der Motor wird dort abgeschaltet, wenn READ ohne Fehler verlaufen ist.

$FCD3-FCD9 **SET ARM POSITION FOR DRIVE IN MEMORY**
Aufruf mit Tracknummer (= Armposition) * 2 in A.
Der Index innerhalb der RWTB-eigenen Tabelle der Armpositionen sämtlicher Units wird berechnet ($FCF1), die momentane Tracknummer (* 2) als „letzte Armposition" für dieses Unit gespeichert. RTS zur aufrufenden Routine.

$FCDA-$FCF0 **IS MOTOR ON?**
Ein Byte wird gelesen, 23 Mikrosekunden gewartet und mit dem nächsten Byte von der Disk (Schieberegister) verglichen. Hat sich das Byte im Schieberegister inzwischen geändert, wird der Motor als „ON" betrachtet. RTS mit „ungleich" (NE) im P-Register.
Ansonsten wird (vorsorglich) die ERR#28 (NO DEVICE CONNECTED) geladen und der Versuch 256 Mal wiederholt. Findet nach dem 256. Versuch immer noch keine Änderung des Schieberegisters statt, RTS mit EQ zur aufrufenden Routine. Das Carry ist undefiniert, die ERR# wird nur beim Aufruf von $F88B berücksichtigt.

$FCF1-$FCFF Berechnet aus der Unitnummer von der Zeropage ($0043) einen Index innerhalb der RWTB-eigenen Tabelle für die Armpositionen nach der Formel: Index = 2 * Slot + Drive. Der Akku wird zwischengespeichert. RTS zur aufrufenden Routine mit Index in X.

$FD00- **WRITE DATAFIELD**
Vorher wurde die „Prenibble"-Routine ($FDF0) ausgeführt, von dort aus wurden auch die Basisadressen für die erste/zweite Hälfte des Quellpuffers berechnet, die Slotadresse modifiziert sowie die Bytes $003B/$003C/$003D/$003F auf der Zeropage gesetzt.

$FD01-$FD09 Test auf WRITE PROTECTED. Wenn PROTECTED, Sprung zu $FDDF.

$FD0C Das letzte „zusätzliche" Byte ($FB00) wird auf $003A gespeichert.

$FD11-$FD34 Fünf „Autosyncs" und der Data Field Header (D5 AA AD) werden via $FDE7 (Write single Byte) geschrieben.

| | |
|---|---|
| $FD36-$FD4E | Die „zusätzlichen" Bytes aus dem RWTB-eigenen Puffer ($FB00-$FB55) werden in absteigender Richtung ($FB55 zuerst) geschrieben. Der Loop endet bei $FB01 EOR $FB00 => Disk. |
| $FD4F | Das vorher auf $003A gespeicherte Byte von $FB00 wird zurückgeladen. |
| $FD51 | Das Y-Register wird mit dem Startindex für den Quellpuffer geladen. Dieser Index ist von PRENIBBLE gesetzt. |
| $FD53-$FD69 | Es werden solange Daten aus dem Quellpuffer in aufsteigender Reihenfolge (Start: Puffer + 00) geschrieben, bis die Seitengrenze erreicht ist. |
| $FD6A | Hat der Quellpuffer auf einer Seitengrenze angefangen, ist $003B von PRENIBBLE auf $00 gesetzt. Sprung zu $FDC0 (Quellpuffer zu Ende, Schreiben der Checksum und des Trailers). |
| $FD6E | Endet der Puffer auf einer Seitengrenze (Start auf $xx01), bleibt nur noch ein Byte zu schreiben übrig. Sprung zu $FDB3. |
| $FD72 | Ansonsten ist es jetzt höchste Zeit, das nächste Byte zu schreiben. Durch Shiften des Endadressen-Index ($003F) wird noch schnell bestimmt, ob der Quellpuffer auf einer geraden oder ungeraden Adresse endet, danach wird das von PRENIBBLE vorübersetzte Byte ($003B), das sich genau auf der Seitengrenze befindet, geschrieben. |
| $FD7F | Das Carry ist dabei erhalten geblieben. Endet der Puffer auf einer ungeraden Adresse, Sprung zu $FD99 mit dem Wert von $003C im Akku (von Prenibble auf Byte $00 EXOR Byte $01 auf der nächsten Seite des Quellpuffers gesetzt).<br>Ansonsten enthält $003C den Wert des Byte $01 auf der zweiten Seite des Quellpuffers und wird in der folgenden Routine mit Byte $02 EXOR'd und geschrieben. |
| $FD81-$FDAD | Der Teil des Quellpuffers, der auf der zweiten Speicherseite steht, wird geschrieben. Ein Loop schreibt jeweils zwei Byte. |
| $FD99 | Einsprung von $FD7F mit bereits EXOR„tem" Byte $00/$01. Das Byte wird geschrieben, danach wird geprüft, ob das Ende des Quellpuffers erreicht ist. Wenn nicht, Loop zu $FD81. Durch diesen Einsprung wird erreicht, daß immer die richtige Anzahl von Bytes geschrieben wird, obwohl der Test auf Pufferende nur alle zwei Bytes stattfindet. |
| $FDAF-$BDB1 | Der Puffer ist zu Ende. Verzögerung um insgesamt 6 T-Zyklen und Sprung zu $FDC0. |
| $FDB3 | Ein Spezialfall: Der Quellpuffer fängt auf $xx01 an und endet damit auf $yy00 der nächsten Seite. Wird nur von $FD70 angesprungen. Das letzte Byte wird geschrieben, weiter mit $FDC0. |
| $FDC0 | Die Checksum wird geschrieben, gefolgt vom Data Field Trailer ($DE AA EB). Für das Schreiben des Trailers wird wiederum $FDE6 benutzt. Die Daten für den Trailer sind in RDTRANS1 versteckt. |
| $FDDF | Exit der WRITE-Routine. Zurückschaltung auf READ und RTS zu $F8F7. Wird auch für den Fehlerfall WRITE PROTECTED benutzt. |
| $FDE6-$FDEF | **WRITE SINGLE BYTE**<br>Hier gibt es verschiedene Einsprünge, je nach benötigtem Timing. Das Byte wird aus dem Akku geschrieben. RTS zur aufrufenden Routine. |

$FDF0- **PRENIBBLE + SET ADDRESS FOR WRITE**

$FDF0-$FE16 Die drei Basisadressen für erstes/zweites/drittes Drittel im folgenden Code werden gesetzt. Da für PRENIBBLE innerhalb jedes Drittels mit einem Index von #$AA begonnen wird, verschieben sich die Basisadressen dementsprechend nach unten.

$FE18 Start des Prenibble-Loops. Es werden alle drei Drittel innerhalb dieses einen Loops bearbeitet.
Aus den Bits 0 und 1 jedes Bytes mit gleichem Index innerhalb eines der drei Drittel ergibt sich zusammen mit drei Übersetzungstabellen ein „zusätzliches Byte", daß im RWTB-eigenen Puffer ($FB00-$FB55) in absteigender Reihenfolge gespeichert wird. Bit 0 und Bit 1 dieses „zusätzlichen" Bytes sind immer 0.

$FE45 Da die WRITE-Routine extrem zeitkritisch ist und der 6502 einen berüchtigten „Haken" hat (1 T-Zyklus mehr bei Seitenüberschreitung durch indexierte Adressen), müssen einige Vorbereitungen getroffen werden.
Der Startindex für den zweiten Teil in WRITE ($FD52) wird auf die niederwertige Quellpuffer-Startadresse gesetzt.
Fängt der Quellpuffer auf einer Seitengrenze an, ergeben sich folgende Werte auf der Zeropage:
$003B:= 00;
$003C:= undefiniert;
$003D:= letztes Byte (AND #$FC) aus dem Puffer, d.h. „Checksum";
$003F:= $FF.

Fängt der Puffer dagegen nicht auf einer Seitengrenze an, ergibt sich:
$003B:= Das Byte mit der Adresse $xxFF auf der ersten Seite des Quellpuffers, EXOR'd mit Byte $yy00 von der zweiten Seite und bereits via WRTRANS übersetzt;
$003C:= Wenn der Puffer auf einer geraden Adresse aufhört: Byte $00 auf der nächsten Seite der Quellpuffers kopiert – ansonsten Byte $00 EXOR Byte $01;
$003D:= letztes Byte (AND #$FC) aus dem Puffer, „Checksum" wie oben;
$003F:= Index zum letzten Byte des Puffers.

$FE77-$FE8B Die Basisadressen der einzelnen WRITE-Loops werden auf den Quellpuffer-Anfang gesetzt.

$FE8C-$FE9A WRITE wird mit der Slotno * 16 des Ziel-Units modifiziert. RTS zu $F89A.

$FE9B-FEBD **IS SAME SLOT?**
Aufruf mit der Nummer des neuen Units im Akku.
Ist der Slot des „neuen" Units derselbe wie der des zuletzt benutzten, RTS. Bit 7 der Unitnummer (Drive) wird hier maskiert.
Ist die Slotnummer des zuletzt benutzten Units 0 (RAMdisk??), ebenfalls RTS.

Ansonsten wird geprüft, ob der Motor des „alten" Units noch an ist ($FCDC). Ist das der Fall, wird eine kurze Zeit gewartet ($FB85) und danach der Test solange wiederholt, bis der Motor aus ist. Der Wert von $FB70 ist dabei unbestimmt (s. „Spezielles").
RTS zur aufrufenden Routine ($F848).

$FEBE-$FEFF nicht benutzt

$FF00-$FF69 Interface zur RAMDisk-RWTB

$FF6A-$FF9A Variablen der RAMDisk-RWTB

$FF9B-$FFFF IRQ/RESET Handler, sind im Kapitel „Interrupts/RESET" besprochen.

# 5.6. Kommentiertes Listing der DISKII-RWTB

Benutzte Speicherstellen auf der Zeropage:

$003A: WRITE: Kopie von $FB00 – letztes der „zusätzlichen" Nibbles ;
$003B: WRITE: $00, wenn Puffer auf Seitengrenze, sonst Byte $00 EOR $01 der nächsten Speicherseite;
$003C: WRITE: Byte $01 der nächsten Speicherseite, wenn der Puffer nicht auf einer Seitengrenze anfängt;
$003D: WRITE: letztes Puffer-Byte für Checksum;
$003E: Slotnummer * 16;
$003F: WRITE: Index zum letzten Byte des Puffers.

Die „offiziellen" Zeropage-Speicherstellen zur RWTB-Parameterübergabe:

$0042 – COMMAND (0 = „Activate"; 1 = READ; 2 = WRITE)
$0043 – Unitnummer (DSSS 0000)
$0044/45 – Ziel/Quelladresse
$0046/47 – Blocknummer, Bereich $00 bis $117 (dez. 279)

– Einziger definierter Einsprung für alle RWTB-Befehle

```
F800-  A5 46      LDA  $46          ; Blockno lo
F802-  A6 47      LDX  $47
F804-  8E 56 FB   STX  $FB56        ; Blockno hi
F807-  F0 07      BEQ  $F810
F809-  CA         DEX
F80A-  D0 2A      BNE  $F836        ; Blockhi = max. 1
F80C-  C9 18      CMP  #$18         ; Blocklo = max. $17 für Blockhi = 1
F80E-  B0 26      BCS  $F836        ; I/O ERROR bei Überschreitung
F810-  A0 05      LDY  #$05
F812-  0A         ASL               ; Blocknummer / 8
F813-  2E 56 FB   ROL  $FB56
F816-  88         DEY
```

```
F817-  D0 F9       BNE   $F812
F819-  0A          ASL
F81A-  90 02       BCC   $F81E          ; Sprung, wenn Sektornummer in 0..3
F81C-  09 10       ORA   #$10           ; sonst + 1
F81E-  4A          LSR                  ; / 16 =>
F81F-  4A          LSR                  ; 0 1 2 3 4 5 6 7
F820-  4A          LSR                  ; - - - - - - - -
F821-  4A          LSR                  ; 0 4 8 C 1 5 9 D
F822-  48          PHA                  ; Speichern der Sektornummer
F823-  20 3A F8    JSR   $F83A          ; READ/WRITE + „00"-Command
F826-  68          PLA
F827-  B0 09       BCS   $F832          ; ERROR
F829-  E6 45       INC   $45            ; Ziel/Quelladresse + $100
F82B-  69 02       ADC   #$02           ; Sektornummer + 2 => zweite Blockhälfte
F82D-  20 3A F8    JSR   $F83A          ; READ/WRITE + „00"-Command
F830-  C6 45       DEC   $45            ; Zurücksetzen der Ziel/Quelladresse
F832-  AD 58 FB    LDA   $FB58          ; ERR# ($00 = kein Fehler)
F835-  60          RTS

F836-  A9 27       LDA   #$27           ; ERR#27: I/O ERROR
F838-  38          SEC
F839-  60          RTS
```

**– READ/WRITE eines Sektors** (+ „00"-Command)
Sektornummer im Akku; Tracknummer: $FB56; Ziel/Quelladresse: $0046/47

```
F83A-  A0 01       LDY   #$01           ; Recalibration Counter auf 1
F83C-  8C 6A FB    STY   $FB6A
F83F-  8D 57 FB    STA   $FB57          ; übersetzte (physikalische) Sektorno
F842-  A5 43       LDA   $43            ; Unitnummer (DSSS 0000)
F844-  29 70       AND   #$70           ; => Slotno X* 16
F846-  85 3E       STA   $3E
F848-  20 9B FE    JSR   $FE9B          ; vergleicht mit zuletzt benutztem Slot,
                                          falls ungleich, WAIT auf „Motor off"
F84B-  20 DA FC    JSR   $FCDA          ; testet, ob Ziel-Drive ON
F84E-  08          PHP                  ; Resultat: NE wenn Motor ON
F84F-  A9 E8       LDA   #$E8           ; die „Motor-on-Delay" Konstante wird
F851-  8D 70 FB    STA   $FB70            immer gesetzt
F854-  A5 43       LDA   $43            ; Unitnummer, diesmal auch Vergleich des
F856-  CD 59 FB    CMP   $FB59            Drives (nicht nur Slotnummer, s. $F848)
F859-  8D 59 FB    STA   $FB59          ; Update des „zuletzt benutzen Units"
F85C-  08          PHP                  ; Z-Flag vom Vergleich
F85D-  0A          ASL                  ; MSB der Unitnummer („Drive")
F85E-  BD 89 C0    LDA   $C089,X        ; MOTOR ON
F861-  90 01       BCC   $F864          ; war Drive 1
F863-  E8          INX                  ; sonst Drive 2
F864-  BD 8A C0    LDA   $C08A,X        ; SELECT DRIVE

F867-  28          PLP                  ; EQ, wenn selber Slot & Drive
F868-  F0 0A       BEQ   $F874
F86A-  28          PLP                  ; ungleich: das „Motor on"-Flag ist dann
F86B-  A0 07       LDY   #$07             ungültig
F86D-  20 85 FB    JSR   $FB85          ; für die Umschaltung (Head Load) muß
F870-  88          DEY                    ein WAIT eingebaut werden
F871-  D0 FA       BNE   $F86D
F873-  08          PHP                  ; „Motor on"-Flag auf „Motor war OFF" (EQ)
```

**– CORRECT DRIVE SELECTED**

```
F874-  A5 42       LDA   $42            ; COMMAND
F876-  F0 06       BEQ   $F87E          ; „00"-Command: keine Armbewegung!
```

```
F878-   AD 56 FB    LDA $FB56          ; gewünschte Tracknummer
F87B-   20 0C F9    JSR   $F90C        ; MYSEEK - Armbewegung zum Ziel-Track

F87E-   28          PLP                ; „Motor on"-Flag
F87F-   D0 0F       BNE   $F890        ; Sprung, wenn der richtige Drive ON war
F881-   A9 01       LDA   #$01
F883-   20 85 FB    JSR   $FB85        ; sonst Motor-on Delay
F886-   AD 70 FB    LDA   $FB70
F889-   30 F6       BMI   $F881
F88B-   20 DA FC    JSR   $FCDA        ; testet, ob der Motor jetzt ON ist
F88E-   F0 5C       BEQ   $F8EC        ; ERROR: Motor ist nicht ON, Exit

F890-   A5 42       LDA   $42          ; COMMAND
F892-   F0 69       BEQ   $F8FD        ; „00"- Drive ist on. Exit via TEST WPROT
F894-   4A          LSR
F895-   B0 03       BCS   $F89A        ; READ
F897-   20 F0 FD    JSR   $FDF0        ; bleibt nur WRITE: Aufruf von PRENIBBLE
```

## – TRY TRACK

```
F89A-   A0 40       LDY   #$40         ; Init des Retry Counters
F89C-   8C 69 FB    STY   $FB69
```

## – RETRY READ

```
F89F-   A6 3E       LDX   $3E          ; Slotno * 16
F8A1-   20 98 FB    JSR   $FB98        ; RDADDR - liest ein Address Field
F8A4-   90 1A       BCC   $F8C0        ; Read ok.

F8A6-   CE 69 FB    DEC   $FB69        ; Retries - 1
F8A9-   10 F4       BPL   $F89F        ; nächster Versuch mit RDADDR
```

## – RECALIBRATE

```
F8AB-   A9 27       LDA   #$27
F8AD-   CE 6A FB    DEC   $FB6A        ; schon einmal rekalibriert?
F8B0-   D0 3A       BNE   $F8EC        ; ja, Exit mit ERR#27: I/O ERROR
F8B2-   AD 5A FB    LDA   $FB5A        ; nein, Speichern der „Trackno gewünscht"
F8B5-   48          PHA
F8B6-   0A          ASL                ; * 2 für SEEKABS
F8B7-   69 10       ADC   #$10         ; => Armbewegung um 8 Tracks nach innen
F8B9-   A0 40       LDY   #$40         ; Re-init der Retries
F8BB-   8C 69 FB    STY   $FB69
F8BE-   D0 0E       BNE   $F8CE        ; „always"

F8C0-   AC 6F FB    LDY   $FB6F        ; „Track gefunden" von RDADDR
F8C3-   CC 5A FB    CPY   $FB5A        ; „Track gewünscht"
F8C6-   F0 0F       BEQ   $F8D7        ; ok. Auf dem richtigen Track
```

## – SETTRK II

```
F8C8-   AD 5A FB    LDA   $FB5A        ; „Track gewünscht"
F8CB-   48          PHA
F8CC-   98          TYA                ; „Track gefunden" von RDADDR
F8CD-   0A          ASL                ; diese Trackno wird als „Track jetzt"
F8CE-   20 D3 FC    JSR   $FCD3          im Armpositions-Speicher gesetzt
F8D1-   68          PLA
F8D2-   20 0C F9    JSR   $F90C        ; neues SEEKABS mit „Trackno gewünscht"
F8D5-   90 C8       BCC   $F89F        ; „always"
```

## – ON CORRECT TRACK

```
F8D7-   AD 6E FB    LDA   $FB6E        ; „Sektorno gefunden" von RDADDR
F8DA-   CD 57 FB    CMP   $FB57        ; „Sektor gewünscht"
F8DD-   D0 C7       BNE   $F8A6        ; Retry – nächster Versuch mit RDADDR
F8DF-   A5 42       LDA   $42          ; COMMAND
F8E1-   4A          LSR
F8E2-   90 10       BCC   $F8F4        ; => $02: WRITE
F8E4-   20 FD FB    JSR   $FBFD        ; READ DATA FIELD + DENIBBLE
F8E7-   B0 BD       BCS   $F8A6        ; Error => Retry
F8E9-   A9 00       LDA   #$00         ; sonst ERR# = $00

F8EB-   D0 38       BNE   $F925        ; $F8EC: „SEC" versteckt
F8ED-   8D 58 FB    STA   $FB58        ; ERR# wird gesetzt ($00 = kein Fehler)
F8F0-   BD 88 C0    LDA   $C088,X      ; MOTOR OFF
F8F3-   60          RTS                ; zu $F826 oder $F830

F8F4-   20 00 FD    JSR   $FD00        ; WRITE DATA FIELD
F8F7-   90 F0       BCC   $F8E9
F8F9-   A9 2B       LDA   #$2B         ; WRITE PROTECTED ist der einzig mögliche
F8FB-   D0 EF       BNE   $F8EC          Fehler in WRITE ($FD00) selber
```

## – „$00"- COMMAND: TEST WRITE PROTECTED

```
F8FD-   A6 3E       LDX   $3E          ; Slotno * 16
F8FF-   BD 8D C0    LDA   $C08D,X
F902-   BD 8E C0    LDA   $C08E,X
F905-   2A          ROL                ; setzt Carry, wenn WRITE PROTECTED
F906-   BD 8C C0    LDA   $C08C,X
F909-   4C F7 F8    JMP   $F8F7
```

## – SEEKABS (MYSEEK)

```
F90C-   0A          ASL                ; Aufruf mit „Track gewünscht" im Akku
F90D-   8D 6F FB    STA   $FB6F        ; „Track gewünscht" * 2
F910-   20 25 F9    JSR   $F925        ; alle Phasen OFF (CLC von „ASL")
F913-   20 F1 FC    JSR   $FCF1        ; X-Reg = Unitnummer * 2 + Drive
F916-   BD 59 FB    LDA   $FB59,X      ; letzte Armposition im Speicher für das
F919-   8D 5A FB    STA   $FB5A          Unit => „Track jetzt"
F91C-   AD 6F FB    LDA   $FB6F        ; nach SEEKABS sollte der „Track gewünscht"
F91F-   9D 59 FB    STA   $FB59,X        erreicht sein: Update der Armposition
F922-   20 33 F9    JSR   $F933        ; SEEKABS

F925-   A0 03       LDY   #$03         ; setzt alle vier Phasen OFF
F927-   98          TYA
F928-   20 8A F9    JSR   $F98A        ; schaltet eine Phase (00 bis 03)
F92B-   88          DEY                ; kein Delay!
F92C-   10 F9       BPL   $F927
F92E-   4E 5A FB    LSR   $FB5A        ; „Track jetzt" / 2 => logische Tracknummer
F931-   18          CLC
F932-   60          RTS                ; => $F913 oder Ende von MYSEEK
```

## – SEEKABS

```
F933-   8D 72 FB    STA   $FB72        ; „Track gewünscht" * 2
F936-   CD 5A FB    CMP   $FB5A        ; „Track jetzt" * 2
F939-   F0 4C       BEQ   $F987   ***  ; Programmierfehler, Carry ist gesetzt
F93B-   A9 00       LDA   #$00         ; sonst Init des Delay-Index auf
F93D-   8D 6B FB    STA   $FB6B          maximale Verzögerung

F940-   AD 5A FB    LDA   $FB5A        ; „Track jetzt" * 2
F943-   8D 71 FB    STA   $FB71
F946-   38          SEC
F947-   ED 72 FB    SBC   $FB72        ; „Track gewünscht" * 2
F94A-   F0 37       BEQ   $F983        ; ok. Abschalten der letzten Phase
```

```
F94C-   B0 07       BCS   $F955        ; „gewünscht" < „jetzt" => nach außen
F94E-   49 FF       EOR   #$FF         ; Vorzeichen der Differenz umdrehen (und -1)
F950-   EE 5A FB    INC   $FB5A        ; und „Track jetzt" + 1
F953-   90 05       BCC   $F95A        ; „always"
F955-   69 FE       ADC   #$FE         ; Bewegung nach außen: Differenz - 1
F957-   CE 5A FB    DEC   $FB5A        ; und „Track jetzt" - 1
F95A-   CD 6B FB    CMP   $FB6B        ; Vergleich Differenz -1 mit Delay-Index:
F95D-   90 03       BCC   $F962          wenn Differenz bereits kleiner, Sprung
F95F-   AD 6B FB    LDA   $FB6B        ; sonst Vergleich: ist der Delay-Index
F962-   C9 09       CMP   #$09           bereits > 8?
F964-   B0 02       BCS   $F968        ; ja, Y-Reg unverändert lassen
F966-   A8          TAY
F967-   38          SEC                ; PHASE ON zum neuen Track hin
F968-   20 87 F9    JSR   $F987
F96B-   B9 73 FB    LDA   $FB73,Y      ; PHASE ON Delay
F96E-   20 85 FB    JSR   $FB85
F971-   AD 71 FB    LDA   $FB71        ; „alter Track"
F974-   18          CLC                ; PHASE OFF vom alten Track weg
F975-   20 8A F9    JSR   $F98A
F978-   B9 7C FB    LDA   $FB7C,Y      ; PHASE OFF Delay
F97B-   20 85 FB    JSR   $FB85
F97E-   EE 6B FB    INC   $FB6B        ; Delay-Index + 1
F981-   D0 BD       BNE   $F940        ; „always"

F983-   20 85 FB    JSR   $FB85        ; Sprung hierher, wenn der „Track gewünscht"
F986-   18          CLC                  erreicht ist: letzte Phase OFF
```

### – ARM MOVER (Phasen-Adressierung) ; SEC: PHASE ON / CLC: PHASE OFF

```
F987-   AD 5A FB    LDA   $FB5A        ; Einsprung für „Track jetzt"
F98A-   29 03       AND   #$03         ; Einsprung für „alten Track"
F98C-   2A          ROL                ; ergibt ungerade Adresse (ON) für SEC
F98D-   05 3E       ORA   $3E          ; Slotno * 16
F98F-   AA          TAX
F990-   BD 80 C0    LDA   $C080,X
F993-   A6 3E       LDX   $3E          ; Reload der Slotno * 16
F995-   60          RTS
```

### – RDTRANS I

```
F996- 00 04 ** ** 08 0C ** 10  14 18 ** ** ** ** ** **  ; wird direkt adressiert:
F9A6- 1C 20 ** ** ** 24 28 2C  30 34 ** ** 38 3C 40 44    die „Disk"-Bytes haben
F9B6- 48 4C ** 50 54 58 5C 60  64 68 ** ** ** ** ** **    Werte zwischen $96 und
F9C6- ** ** ** ** ** 6C ** 70  74 78 ** ** ** 7C ** **    $FF, die Übersetzung
F9D6- 80 84 ** 88 8C 90 94 98  9C A0 ** ** ** ** ** A4    erfolgt mit Basis
F9E6- A8 AC ** B0 B4 B8 BC C0  C4 C8 ** ** CC D0 D4 D8    $F900 + Index, wobei
F9F6- DC E0 ** E4 E8 EC F0 F4  F8 FC                      der Index das gelesene
                                                          „Disk"-Byte selber ist.
```

Andere Kurztabellen, in RDTRANS I versteckt:

```
F9A0- 00 80 40 C0                  ; von PRENIBBLE benutzt
F9C0- 00 20 10 30
F9E0- 00 08 04 0C

F9C4- DE AA EB                     ; Data Field Trailer
```

**– RDTRANS II/1 –** Basis ist $FA00

```
FA00- 00 ** ** ** 02 ** ** **  01 ** ** ** 03 ** ** **  ; adressiert durch die
FA10- 00 ** ** ** 02 ** ** **  01 ** ** ** 03 ** ** **    „zusätzlichen" Bytes.
FA20- 00 ** ** ** 02 ** ** **  01 ** ** ** 03 ** ** **    Enthält Bit 0/1 für das
FA30- 00 ** ** ** 02 ** ** **  01 ** ** ** 03 ** ** **    erste Drittel des
FA40- 00 ** ** ** 02 ** ** **  01 ** ** ** 03 ** ** **    Datenpuffers für READ.
FA50- 00 ** ** ** 02 ** ** **  01 ** ** ** 03 ** ** **
FA60- 00 ** ** ** 02 ** ** **  01 ** ** ** 03 ** ** **
FA70- 00 ** ** ** 02 ** ** **  01 ** ** ** 03 ** ** **
FA80- 00 ** ** ** 02 ** ** **  01 ** ** ** 03 ** ** **
FA90- 00 ** ** ** 02 ** ** **  01 ** ** ** 03 ** ** **
FAA0- 00 ** ** ** 02 ** ** **  01 ** ** ** 03 ** ** **
FAB0- 00 ** ** ** 02 ** ** **  01 ** ** ** 03 ** ** **
FAC0- 00 ** ** ** 02 ** ** **  01 ** ** ** 03 ** ** **
FAD0- 00 ** ** ** 02 ** ** **  01 ** ** ** 03 ** ** **
FAE0- 00 ** ** ** 02 ** ** **  01 ** ** ** 03 ** ** **
FAF0- 00 ** ** ** 02 ** ** **  01 ** ** ** 03 ** ** **
```

**– RDTRANS II/2 –** Basis ist $FA01

```
FA00- ** 00 ** ** ** 00 ** **  ** 00 ** ** ** 00 ** **  ; enthält Bit 0/1 für
FA10- ** 02 ** ** ** 02 ** **  ** 02 ** ** ** 02 ** **    das zweite Drittel des
FA20- ** 01 ** ** ** 01 ** **  ** 01 ** ** ** 01 ** **    READ-Datenpuffers
FA30- ** 03 ** ** ** 03 ** **  ** 03 ** ** ** 03 ** **
FA40- ** 00 ** ** ** 00 ** **  ** 00 ** ** ** 00 ** **
FA50- ** 02 ** ** ** 02 ** **  ** 02 ** ** ** 02 ** **
FA60- ** 01 ** ** ** 01 ** **  ** 01 ** ** ** 01 ** **
FA70- ** 03 ** ** ** 03 ** **  ** 03 ** ** ** 03 ** **
FA80- ** 00 ** ** ** 00 ** **  ** 00 ** ** ** 00 ** **
FA90- ** 02 ** ** ** 02 ** **  ** 02 ** ** ** 02 ** **
FAC0- ** 00 ** ** ** 00 ** **  ** 00 ** ** ** 00 ** **
FAD0- ** 02 ** ** ** 02 ** **  ** 02 ** ** ** 02 ** **
FAE0- ** 01 ** ** ** 01 ** **  ** 01 ** ** ** 01 ** **
FAF0- ** 03 ** ** ** 03 ** **  ** 03 ** ** ** 03 ** **
```

**– RDTRANS II/3 –** Basis ist $FA02

```
FA00- ** ** 00 ** ** ** 00 **  ** ** 00 ** ** ** 00 **  ; enthält Bit 0/1 für das
FA10- ** ** 00 ** ** ** 00 **  ** ** 00 ** ** ** 00 **    dritte Drittel des
FA20- ** ** 00 ** ** ** 00 **  ** ** 00 ** ** ** 00 **    READ-Datenpuffers
FA30- ** ** 00 ** ** ** 00 **  ** ** 00 ** ** ** 00 **
FA40- ** ** 02 ** ** ** 02 **  ** ** 02 ** ** ** 02 **
FA50- ** ** 02 ** ** ** 02 **  ** ** 02 ** ** ** 02 **
FA60- ** ** 02 ** ** ** 02 **  ** ** 02 ** ** ** 02 **
FA70- ** ** 02 ** ** ** 02 **  ** ** 02 ** ** ** 02 **
FA80- ** ** 01 ** ** ** 01 **  ** ** 01 ** ** ** 01 **
FA90- ** ** 01 ** ** ** 01 **  ** ** 01 ** ** ** 01 **
FAA0- ** ** 01 ** ** ** 01 **  ** ** 01 ** ** ** 01 **
FAB0- ** ** 01 ** ** ** 01 **  ** ** 01 ** ** ** 01 **
FAC0- ** ** 03 ** ** ** 03 **  ** ** 03 ** ** ** 03 **
FAD0- ** ** 03 ** ** ** 03 **  ** ** 03 ** ** ** 03 **
FAE0- ** ** 03 ** ** ** 03 **  ** ** 03 ** ** ** 03 **
FAF0- ** ** 03 ** ** ** 03 **  ** ** 03 ** ** ** 03 **
```

**– WRTRANS TABLE –** Basis ist $FA03

```
FA00- ** ** ** 96 ** ** ** 97  ** ** ** 9A ** ** ** 9B  ; enthält die gültigen
FA10- ** ** ** 9D ** ** ** 9E  ** ** ** 9F ** ** ** A6    „Disk"-Bytes für WRITE
FA20- ** ** ** A7 ** ** ** AB  ** ** ** AC ** ** ** AD
FA30- ** ** ** AE ** ** ** AF  ** ** ** B2 ** ** ** B3
FA40- ** ** ** B4 ** ** ** B5  ** ** ** B6 ** ** ** B7
FA50- ** ** ** B9 ** ** ** BA  ** ** ** BB ** ** ** BC
FA60- ** ** ** BD ** ** ** BE  ** ** ** BF ** ** ** CB
FA70- ** ** ** CD ** ** ** CE  ** ** ** CF ** ** ** D3
FA80- ** ** ** D6 ** ** ** D7  ** ** ** D9 ** ** ** DA
FA90- ** ** ** DB ** ** ** DC  ** ** ** DD ** ** ** DE
FAA0- ** ** ** DF ** ** ** E5  ** ** ** E6 ** ** ** E7
FAB0- ** ** ** E9 ** ** ** EA  ** ** ** EB ** ** ** EC
FAC0- ** ** ** ED ** ** ** EE  ** ** ** EF ** ** ** F2
FAD0- ** ** ** F3 ** ** ** F4  ** ** ** F5 ** ** ** F6
FAE0- ** ** ** F7 ** ** ** F9  ** ** ** FA ** ** ** FB
FAF0- ** ** ** FC ** ** ** FD  ** ** ** FE ** ** ** FF
```

**– PUFFER FÜR PRENIBBLE**

Enthält die 2 LSBs jedes Bytes aus dem Datenpuffer in absteigender Reihenfolge:

FB00 - FB55
; $FB55: $AC_0\ AC_1$ $56_0\ 56_1$ $00_0\ 00_1$ (0 0)
; $FB54: $AD_0\ AD_1$ $57_0\ 57_1$ $01_0\ 01_1$ (0 0)
; .....
; $FB00: xx xx $AB_0\ AB_1$ $55_0\ 55_1$ (0 0)

**TEMPORARIES & VARIABLEN DER RWTB**

```
*  $FB56:  Trackno (Block/8)     $FB57: Sektornummer gewünscht (physikalisch)
*  $FB58:  ERR#                  $FB59: Zuletzt benutztes Unit
*  $FB5A:  „Track jetzt"
*  $FB5B: ** ** ** ** ** ** ** ; Speicher für die Armpositionen: D1/D2
* -$FB68                         in den Slots 1..7
*  $FB69: Retry count (#$40)
*  $FB6A: Recal count (#1)
*
*  - Speicherstellen für RDADDR
*  $FB6B: Reseek count (#$FC => 4)
*  $FB6C: Scratch für even/odd  $FB6D: Checksum
*  $FB6E: „Sektorno gefunden"   $FB6F: „Trackno gefunden"
*
*  $FB70: Delay Counter für DRVON/DRVOFF. Wird auch vom Arm Mover erhöht!
*  $FB71: „Track jetzt" * 2 (SEEKABS)
*  $FB72: „Track gewünscht" * 2 (SEEKABS)
*
*  FB73- 01 30 28 24 20 1E 1D 1C 1C          ; PHASE ON DELAY
*  FB7C- 70 2C 26 22 1F 1E 1D 1C 1C          ; PHASE OFF DELAY
```

**– ARM MOVER DELAY –** auch für MOTOR ON/OFF und Head Load benutzt

```
FB85-   A2 11       LDX   #$11
FB87-   CA          DEX
FB88-   D0 FD       BNE   $FB87
FB8A-   EE 6F FB    INC   $FB6F
FB8D-   D0 03       BNE   $FB92
FB8F-   EE 70 FB    INC   $FB70
```

```
FB92-  38          SEC
FB93-  E9 01       SBC   #$01
FB95-  D0 EE       BNE   $FB85
FB97-  60          RTS
```

## – RDADDR

```
FB98-  A0 FC       LDY   #$FC          ; Init der Re-seeks auf 4
FB9A-  8C 6B FB    STY   $FB6B
FB9D-  C8          INY
FB9E-  D0 05       BNE   $FBA5
FBA0-  EE 6B FB    INC   $FB6B
FBA3-  F0 56       BEQ   $FBFB         ; SEC/RTS
FBA5-  BD 8C C0    LDA   $C08C,X
FBA8-  10 FB       BPL   $FBA5
FBAA-  C9 D5       CMP   #$D5          ; $D5
FBAC-  D0 EF       BNE   $FB9D
FBAE-  EA          NOP
FBAF-  BD 8C C0    LDA   $C08C,X
FBB2-  10 FB       BPL   $FBAF
FBB4-  C9 AA       CMP   #$AA          ; $AA
FBB6-  D0 F2       BNE   $FBAA
FBB8-  A0 03       LDY   #$03
FBBA-  BD 8C C0    LDA   $C08C,X
FBBD-  10 FB       BPL   $FBBA
FBBF-  C9 96       CMP   #$96          ; $96
FBC1-  D0 E7       BNE   $FBAA
FBC3-  78          SEI                 ; ********** ! **********
FBC4-  A9 00       LDA   #$00
FBC6-  8D 6C FB    STA   $FB6C         ; Checksum – „Mitläufer"
FBC9-  BD 8C C0    LDA   $C08C,X
FBCC-  10 FB       BPL   $FBC9
FBCE-  2A          ROL                 ; erste Hälfte des Bytes
FBCF-  8D 6B FB    STA   $FB6B
FBD2-  BD 8C C0    LDA   $C08C,X
FBD5-  10 FB       BPL   $FBD2
FBD7-  2D 6B FB    AND   $FB6B         ; erste + zweite Hälfte
FBDA-  99 6D FB    STA   $FB6D,Y       ; $BF6D: CHECK/SECNO/TRACKNO/VOLNO
FBDD-  4D 6C FB    EOR   $FB6C
FBE0-  88          DEY
FBE1-  10 E3       BPL   $FBC6
FBE3-  A8          TAY                 ; enthält die EOR-Kette
FBE4-  D0 15       BNE   $FBFB         ; ERROR: zusammen mit Checksum <> 00
FBE6-  BD 8C C0    LDA   $C08C,X
FBE9-  10 FB       BPL   $FBE6
FBEB-  C9 DE       CMP   #$DE          ; $DE
FBED-  D0 0C       BNE   $FBFB
FBEF-  EA          NOP
FBF0-  BD 8C C0    LDA   $C08C,X
FBF3-  10 FB       BPL   $FBF0
FBF5-  C9 AA       CMP   #$AA          ; $AA
FBF7-  D0 02       BNE   $FBFB
FBF9-  18          CLC
FBFA-  60          RTS
FBFB-  38          SEC                 ; ERROR Exit
FBFC-  60          RTS
```

## – READ DATAFIELD & DENIBBLE

```
FBFD-  8A          TXA                 ; Slotno * 16
FBFE-  09 8C       ORA   #$8C
FC00-  8D 5A FC    STA   $FC5A         ; der Code von READ wird mit der Slotno
FC03-  8D 73 FC    STA   $FC73           modifiziert
```

```
FC06-  8D 89 FC     STA   $FC89
FC09-  8D 9D FC     STA   $FC9D
FC0C-  8D B2 FC     STA   $FCB2
FC0F-  A5 44        LDA   $44          ; der Code von READ wird mit der
FC11-  A4 45        LDY   $45            Zieladresse modifiziert
FC13-  8D AF FC     STA   $FCAF
FC16-  8C B0 FC     STY   $FCB0        ; Basis für das letzte Drittel
FC19-  38           SEC
FC1A-  E9 54        SBC   #$54
FC1C-  B0 01        BCS   $FC1F
FC1E-  88           DEY
FC1F-  8D 97 FC     STA   $FC97
FC22-  8C 98 FC     STY   $FC98        ; Basis für das zweite Drittel
FC25-  38           SEC
FC26-  E9 57        SBC   #$57
FC28-  B0 01        BCS   $FC2B
FC2A-  88           DEY
FC2B-  8D 70 FC     STA   $FC70
FC2E-  8C 71 FC     STY   $FC71        ; Basis für das erste Drittel

FC31-  A0 20        LDY   #$20         ; Start des eigentlichen READ
FC33-  88           DEY
FC34-  F0 37        BEQ   $FC6D        ; SEC / RTS
FC36-  BD 8C C0     LDA   $C08C,X
FC39-  10 FB        BPL   $FC36
FC3B-  49 D5        EOR   #$D5         ; $D5
FC3D-  D0 F4        BNE   $FC33
FC3F-  EA           NOP
FC40-  BD 8C C0     LDA   $C08C,X
FC43-  10 FB        BPL   $FC40
FC45-  C9 AA        CMP   #$AA         ; $AA
FC47-  D0 F2        BNE   $FC3B
FC49-  EA           NOP
FC4A-  BD 8C C0     LDA   $C08C,X
FC4D-  10 FB        BPL   $FC4A
FC4F-  C9 AD        CMP   #$AD         ; $AD
FC51-  D0 E8        BNE   $FC3B

FC53-  A0 AA        LDY   #$AA
FC55-  A9 00        LDA   #$00
FC57-  85 3A        STA   $3A          ; $003A: Temp für die EOR-Kette (Checksum)
FC59-  AE ** C0     LDX   $C0EC
FC5C-  10 FB        BPL   $FC59
FC5E-  BD 00 F9     LDA   $F900,X      ; RDTRANS I: Start auf $F996
FC61-  99 56 FA     STA   $FA56,Y      ; „zusätzliche" Nibbles werden „abwärts"
FC64-  45 3A        EOR   $3A            und immer auf $FB55..$FB00 gelesen
FC66-  C8           INY
FC67-  D0 EE        BNE   $FC57
FC69-  A0 AA        LDY   #$AA
FC6B-  D0 05        BNE   $FC72        ; „always"

FC6D-  38           SEC                ; ERROR Exit: $20 Retries heruntergezählt
FC6E-  60           RTS

FC6F-  99 ** **     STA   $0F56,Y      ; Beispiel $1000: Basis Pufferstart-$AB
FC72-  AE ** C0     LDX   $C0EC          => Puffer + $00 .. Puffer + $54
FC75-  10 FB        BPL   $FC72
FC77-  5D 00 F9     EOR   $F900,X
FC7A-  BE 56 FA     LDX   $FA56,Y
FC7D-  5D 00 FA     EOR   $FA00,X      ; RDTRANS II/1 - dasselbe wie ein „ORA"
FC80-  C8           INY
FC81-  D0 EC        BNE   $FC6F
FC83-  48        ** PHA                ; das letzte Byte des ersten Drittels
FC84-  29 FC        AND   #$FC           wird später in den Zielpuffer
```

```
FC86-   A0 AA       LDY   #$AA           gespeichert: die Zieladresse ist die
FC88-   AE ** C0    LDX   $C0EC          Basis des ersten Drittels + $100
FC8B-   10 FB       BPL   $FC88
FC8D-   5D 00 F9    EOR   $F900,X
FC90-   BE 56 FA    LDX   $FA56,Y
FC93-   5D 01 FA    EOR   $FA01,X
FC96-   99 ** **    STA   $0FAC,Y      ; Beispiel $1000: Basis Pufferstart - $54
FC99-   C8          INY                ; => Puffer + $56 .. Puffer + $AB
FC9A-   D0 EC       BNE   $FC88
FC9C-   AE ** C0    LDX   $C0EC
FC9F-   10 FB       BPL   $FC9C
FCA1-   29 FC       AND   #$FC
FCA3-   A0 AC       LDY   #$AC
FCA5-   5D 00 F9    EOR   $F900,X
FCA8-   BE 54 FA    LDX   $FA54,Y
FCAB-   5D 02 FA    EOR   $FA02,X
FCAE-   99 ** **    STA   $1000,Y      ; Beispiel $1000: Basis Pufferstart - $00
FCB1-   AE ** C0    LDX   $C0EC        ; => Puffer + $AC .. Puffer + $FF
FCB4-   10 FB       BPL   $FCB1
FCB6-   C8          INY
FCB7-   D0 EC       BNE   $FCA5
FCB9-   29 FC       AND   #$FC         ; Checksum
FCBB-   5D 00 F9    EOR   $F900,X
FCBE-   D0 0C       BNE   $FCCC        ; zusammen mit EOR-Kette nicht $00: ERROR

FCC0-   A6 3E       LDX   $3E          ; Reload der Slotno * 16
FCC2-   BD 8C C0    LDA   $C08C,X
FCC5-   10 FB       BPL   $FCC2
FCC7-   C9 DE       CMP   #$DE         ; $DE
FCC9-   18          CLC
FCCA-   F0 01       BEQ   $FCCD        ; das folgende $AA wird nicht geprüft!
FCCC-   38          SEC
FCCD-   68      **  PLA                ; letztes Byte des ersten Drittels
FCCE-   A0 55       LDY   #$55
FCD0-   91 44       STA   ($44),Y      ; => $56. Byte des Zielpuffers
FCD2-   60          RTS
```

– Setzt die Armposition für ein Unit im Speicher

```
FCD3-   20 F1 FC    JSR   $FCF1        ; berechnet Slotno * 2 + Drive
FCD6-   9D 59 FB    STA   $FB59,X      ; Eintrag in den Armpositions-Speicher
FCD9-   60          RTS
```

**– IS MOTOR ON?**

```
FCDA-   A6 3E       LDX   $3E          ; Slotno * 16
FCDC-   A0 00       LDY   #$00         ; Einsprung mit Slotno („alter" Drive)
FCDE-   BD 8C C0    LDA   $C08C,X
FCE1-   20 F0 FC    JSR   $FCF0        ; RTS – 12 Mikrosekunden Delay
FCE4-   48          PHA
FCE5-   68          PLA
FCE6-   DD 8C C0    CMP   $C08C,X      ; insgesamt 23 Mikrosekunden zwischen zwei
FCE9-   D0 05       BNE   $FCF0          READs, wenn sich die Daten am Lesekopf
FCEB-   A9 28       LDA   #$28           ändern, wird NE returnt („Motor ON")
FCED-   88          DEY
FCEE-   D0 EE       BNE   $FCDE        ; sonst EQ und vorsorglich ERR=$28
FCF0-   60          RTS
```

– Berechnet Slot * 2 + Drive als Index für den Armpositions-Speicher

```
FCF1-   48          PHA                ; enthält Tracknummer * 2
FCF2-   A5 43       LDA   $43          ; Unitnummer: DSSS 0000
```

```
FCF4-  4A          LSR
FCF5-  4A          LSR
FCF6-  4A          LSR
FCF7-  4A          LSR
FCF8-  C9 08       CMP   #$08          ; setzt Carry, wenn Drive 2
FCFA-  29 07       AND   #$07
FCFC-  2A          ROL                 ; „addiert" das Carry als LSB
FCFD-  AA          TAX
FCFE-  68          PLA
FCFF-  60          RTS
```

## – WRITE DATA FIELD

```
FD00-  38          SEC
FD01-  BD 8D C0    LDA   $C08D,X
FD04-  BD 8E C0    LDA   $C08E,X
FD07-  10 03       BPL   $FD0C         ; Sprung, wenn nicht WRITE PROTECTED
FD09-  4C DF FD    JMP   $FDDF         ; ERROR: Exit mit SEC
FD0C-  AD 00 FB    LDA   $FB00
FD0F-  85 3A       STA   $3A           ; letztes Byte der „zusätzlichen" - Timing!
FD11-  A9 FF       LDA   #$FF
FD13-  9D 8F C0    STA   $C08F,X       ; Setzen von WRITE
FD16-  1D 8C C0    ORA   $C08C,X       ; Strobe und erstes Autosync
FD19-  A0 04       LDY   #$04
FD1B-  EA          NOP
FD1C-  48          PHA
FD1D-  68          PLA
FD1E-  48          PHA                 ; insgesamt 40 Mikrosek. zwischen 2 Strobes
FD1F-  68          PLA
FD20-  20 E7 FD    JSR   $FDE7         ; WRITE single Byte
FD23-  88          DEY
FD24-  D0 F8       BNE   $FD1E
FD26-  A9 D5       LDA   #$D5          ; $D5
FD28-  20 E6 FD    JSR   $FDE6
FD2B-  A9 AA       LDA   #$AA          ; $AA
FD2D-  20 E6 FD    JSR   $FDE6
FD30-  A9 AD       LDA   #$AD          ; $AD
FD32-  20 E6 FD    JSR   $FDE6
FD35-  98          TYA                 ; = 00

FD36-  A0 56       LDY   #$56          ; die „zusätzlichen" Nibbles werden
FD38-  D0 03       BNE   $FD3D           „abwärts" geschrieben: $FB55 .. $FB00
FD3A-  B9 00 FB    LDA   $FB00,Y         #00   EOR $FB55 => WRTRANS => Byte 0
FD3D-  59 FF FA    EOR   $FAFF,Y         $FB55 EOR $FB54 => WRTRANS => Byte 1
FD40-  AA          TAX
FD41-  BD 03 FA    LDA   $FA03,X
FD44-  A6 3E       LDX   $3E
FD46-  9D 8D C0    STA   $C08D,X       ; „Load Data Latch"
FD49-  BD 8C C0    LDA   $C08C,X       ; Strobe: „Abschicken" des Bytes
FD4C-  88          DEY
FD4D-  D0 EB       BNE   $FD3A         ; Loop, bis $FB00 erreicht ist

FD4F-  A5 3A       LDA   $3A           ; letztes „zusätzliches" Byte ($FB00)
FD51-  A0 **       LDY   #$00          ; Startindex durch PRENIBBLE gesetzt
FD53-  59 ** **    EOR   $1000,Y       ; Pufferstart durch PRENIBBLE gesetzt
FD56-  29 FC       AND   #$FC          ; Maskieren von Bit 0/1
FD58-  AA          TAX
FD59-  BD 03 FA    LDA   $FA03,X       ; WRTRANS
FD5C-  A2 **       LDX   #$60          ; Slotno * 16 von PRENIBBLE gesetzt
FD5E-  9D 8D C0    STA   $C08D,X       ; WRITE des Datenteils in aufsteigender
FD61-  BD 8C C0    LDA   $C08C,X         Reihenfolge:
FD64-  B9 ** **    LDA   $1000,Y         Bstart+00 EOR Bstart+01 => WRTRANS
FD67-  C8          INY                   Bstart+01 EOR Bstart+02 => WRTRANS
```

```
FD68-   D0 E9       BNE   $FD53        ; Loop bis Y=00. Nicht unbedingt
                                          das Pufferende!
FD6A-   A5 3B       LDA   $3B          ; BUFL0 oder Byte auf Seitengrenze!
FD6C-   F0 52       BEQ   $FDC0        ; 00 => Bstart auf Seitengrenze => Ende
FD6E-   A5 3F       LDA   $3F          ; Index zum letzten Pufferbyte
FD70-   F0 41       BEQ   $FDB3        ; wenn Bufend auf 00, WRITE letztes Byte
FD72-   4A          LSR                ; sonst Test, ob Bufend auf EVEN/ODD
FD73-   A5 3B       LDA   $3B          ; übersetztes Byte auf Seitengrenze
FD75-   9D 8D C0    STA   $C08D,X      ; WRITE - wird höchste Zeit!
FD78-   BD 8C C0    LDA   $C08C,X        das Carry bleibt dabei unverändert
FD7B-   A5 3C       LDA   $3C          ; erstes Byte auf der nächsten Seite:
FD7D-   EA          NOP                  wenn Bufend auf ODD, dann ist dieses
FD7E-   C8          INY                  Byte bereits EOR'd
FD7F-   B0 18       BCS   $FD99        ; Carry von EVEN/ODD ($FD72)

FD81-   59 ** **    EOR   $1100,Y      ; nächste Speicherseite des Puffers
FD84-   29 FC       AND   #$FC
FD86-   AA          TAX                ; EVEN (EOR ODD)
FD87-   BD 03 FA    LDA   $FA03,X
FD8A-   A2 **       LDX   #$60
FD8C-   9D 8D C0    STA   $C08D,X
FD8F-   BD 8C C0    LDA   $C08C,X
FD92-   B9 ** **    LDA   $1100,Y      ; ODD
FD95-   C8          INY
FD96-   59 ** **    EOR   $1100,Y      ; EOR EVEN
FD99-   C4 3F       CPY   $3F          ; Index zum letzten Pufferbyte
FD9B-   29 FC       AND   #$FC
FD9D-   AA          TAX
FD9E-   BD 03 FA    LDA   $FA03,X
FDA1-   A2 **       LDX   #$60
FDA3-   9D 8D C0    STA   $C08D,X      ; WRITE: Carry von „CPY $3F" bleibt!
FDA6-   BD 8C C0    LDA   $C08C,X
FDA9-   B9 ** **    LDA   $1100,Y      ; nächstes EVEN Byte
FDAC-   C8          INY
FDAD-   90 D2       BCC   $FD81        ; Carry von „CPY $3F", Loop bis Bufend

FDAF-   B0 00       BCS   $FDB1        ; Puffer geschrieben: 6 Mikrosekunden
FDB1-   B0 0D       BCS   $FDC0          Delay, danach Checksum + Trailer

FDB3-   AD 3B 00    LDA   $003B        ; Spezialfall: Pufferstart auf $XX01, d.h.
FDB6-   9D 8D C0    STA   $C08D,X        Bufend auf $YY00. WRITE des letzten Bytes
FDB9-   BD 8C C0    LDA   $C08C,X
FDBC-   48          PHA
FDBD-   68          PLA
FDBE-   48          PHA
FDBF-   68          PLA

FDC0-   A6 3D       LDX   $3D          ; Checksum
FDC2-   BD 03 FA    LDA   $FA03,X
FDC5-   A2 **       LDX   #$60
FDC7-   9D 8D C0    STA   $C08D,X
FDCA-   BD 8C C0    LDA   $C08C,X
FDCD-   A0 00       LDY   #$00
FDCF-   48          PHA
FDD0-   68          PLA
FDD1-   EA          NOP
FDD2-   EA          NOP
FDD3-   B9 C4 F9    LDA   $F9C4,Y      ; Trailer: $DE AA EB, in RDTRANS I
FDD6-   20 E9 FD    JSR   $FDE9        ; WRITE single Byte
FDD9-   C8          INY
FDDA-   C0 04       CPY   #$04
FDDC-   D0 F5       BNE   $FDD3
FDDE-   18          CLC
FDDF-   BD 8E C0    LDA   $C08E,X      ; Setzen von READ
```

```
FDE2-   BD 8C C0    LDA   $C08C,X
FDE5-   60          RTS
```

### – WRITE single Byte

```
FDE6-   18          CLC             ; diverse Einsprünge, je nach Timing
FDE7-   48          PHA
FDE8-   68          PLA
FDE9-   9D 8D C0    STA   $C08D,X   ; „Load Data Latch"
FDEC-   1D 8C C0    ORA   $C08C,X   ; Strobe - nur bei $FF bleibt Akku gleich
FDEF-   60          RTS
```

### – PRENIBBLE (setzt auch Pufferadressen und Slotno für WRITE)

```
FDF0-   A5 44       LDA   $44       ; Quellpuffer-Startadresse
FDF2-   A4 45       LDY   $45
FDF4-   18          CLC             ; Modifikation von PRENIBBLE selber:
FDF5-   69 02       ADC   #$02
FDF7-   90 01       BCC   $FDFA
FDF9-   C8          INY
FDFA-   8D 30 FE    STA   $FE30     ; Pufferstart + $02 => Puffer + $AC
FDFD-   8C 31 FE    STY   $FE31       (letztes Drittel)
FE00-   38          SEC
FE01-   E9 56       SBC   #$56
FE03-   B0 01       BCS   $FE06
FE05-   88          DEY
FE06-   8D 25 FE    STA   $FE25     ; Pufferstart -$54 => Puffer + $56
FE09-   8C 26 FE    STY   $FE26       (zweites Drittel)
FE0C-   38          SEC
FE0D-   E9 56       SBC   #$56
FE0F-   B0 01       BCS   $FE12
FE11-   88          DEY
FE12-   8D 1B FE    STA   $FE1B     ; Pufferstart -$AA => Puffer + 00
FE15-   8C 1C FE    STY   $FE1C       (erstes Drittel)

FE18-   A0 AA       LDY   #$AA
FE1A-   B9 ** **    LDA   $0F56,Y   ; Puffer + 00 (Beispiel $1000)
FE1D-   29 03       AND   #$03      ; .. .. .. .. 00/0 00/1 (00 00)
FE1F-   AA          TAX
FE20-   BD E0 F9    LDA   $F9E0,X   ; 00 08 04 0C
FE23-   48          PHA
FE24-   B9 ** **    LDA   $0FAC,Y   ; Puffer + $56 (Beispiel $1000)
FE27-   29 03       AND   #$039     ; .. .. 56/0 56/1 .. .. (00 00)
FE29-   AA          TAX
FE2A-   68          PLA
FE2B-   1D C0 F9    ORA   $F9C0,X   ; 00 20 10 30
FE2E-   48          PHA
FE2F-   B9 ** **    LDA   $1002,Y   ; Puffer + $AC (Beispiel $1000)
FE32-   29 03       AND   #$03      ; AC/0 AC/1 .. .. .. .. (00 00)
FE34-   AA          TAX
FE35-   68          PLA
FE36-   1D A0 F9    ORA   $F9A0,X   ; 00 80 40 C0
FE39-   48          PHA
FE3A-   98          TYA
FE3B-   49 FF       EOR   #$FF      ; Speichern der „zusätzlichen" Nibbles
FE3D-   AA          TAX               in umgekehrter Reihenfolge (abwärts)
FE3E-   68          PLA
FE3F-   9D 00 FB    STA   $FB00,X   ; in den Bereich $FB55..$FB00
FE42-   C8          INY               Dieser Bereich bleibt immer gleich!
FE43-   D0 D5       BNE   $FE1A     ; Loop bis Pufferende
```

– Modification von WRITE mit Pufferstart + Slotno, „Vor-Übersetzungen"

```
FE45-   A4 44       LDY   $44          ; Pufferstart lo
FE47-   88          DEY
FE48-   84 3F       STY   $3F          ; Index zum letzten Pufferbyte (DEY = + $FF)
FE4A-   A5 44       LDA   $44
FE4C-   8D 52 FD    STA   $FD52        ; Startindex im Block der „normalen" Bytes
FE4F-   F0 0E       BEQ   $FE5F        ; Sprung, wenn Start auf Seitengrenze
FE51-   49 FF       EOR   #$FF         ; sonst wird der Index zum Byte $xxFF auf
FE53-   A8          TAY                  der ersten Pufferseite berechnet
FE54-   B1 44       LDA   ($44),Y      ; Adresse: $xFF
FE56-   C8          INY
FE57-   51 44       EOR   ($44),Y      ; muß hier übersetzt werden (Timing)
FE59-   29 FC       AND   #$FC         ; $xxFF EOR $yy00 von der zweiten Seite
FE5B-   AA          TAX
FE5C-   BD 03 FA    LDA   $FA03,X      ; WRTRANS
FE5F-   85 3B       STA   $3B          ; $3B = 00, wenn Sprung hierher von $FE4F
FE61-   F0 0C       BEQ   $FE6F        ; Sprung, wenn von $FE4F gekommen
FE63-   A5 3F       LDA   $3F          ; Index zum letzten Pufferbyte auf der
FE65-   4A          LSR                  zweiten Seite (EVEN/ODD-Ende des Puffers)
FE66-   B1 44       LDA   ($44),Y      ; Adresse: Byte $yy01 auf der 2. Seite
FE68-   90 03       BCC   $FE6D        ; Sprung wenn Bufend EVEN (C von „LSR")
FE6A-   C8          INY                ; sonst EOR mit Byte $yy02 der 2.Seite
FE6B-   51 44       EOR   ($44),Y        => Bufend EVEN: Byte $yy01 von der 2.Seite
FE6D-   85 3C       STA   $3C            => Bufend ODD: $yy01 EOR $yy02 (2. Seite)

FE6F-   A0 FF       LDY   #$FF
FE71-   B1 44       LDA   ($44),Y      ; Adresse: letztes Pufferbyte
FE73-   29 FC       AND   #$FC
FE75-   85 3D       STA   $3D          ; CHECKSUM

FE77-   A4 45       LDY   $45          ; Modifikation von WRITE mit Pufferhi
FE79-   8C 55 FD    STY   $FD55
FE7C-   8C 66 FD    STY   $FD66
FE7F-   C8          INY                ; und Pufferhi + 1, wenn Pufferstart nicht
FE80-   8C 83 FD    STY   $FD83        ; auf $xx00 oder $xx01
FE83-   8C 94 FD    STY   $FD94
FE86-   8C 98 FD    STY   $FD98
FE89-   8C AB FD    STY   $FDAB

FE8C-   A6 3E       LDX   $3E          ; Modifikation von WRITE mit Slotno * 16
FE8E-   8E 5D FD    STX   $FD5D
FE91-   8E 8B FD    STX   $FD8B
FE94-   8E A2 FD    STX   $FDA2
FE97-   8E C6 FD    STX   $FDC6
FE9A-   60          RTS
```

## – IS SAME SLOT?

```
FE9B-   4D 59 FB    EOR   $FB59        ; Aufruf mit Nummer des neuen Units
FE9E-   0A          ASL                ; D1/D2 ist unwichtig
FE9F-   F0 1C       BEQ   $FEBD        ; derselbe Slot: RTS
FEA1-   A9 01       LDA   #$01
FEA3-   8D 70 FB    STA   $FB70        ; Counter für Anzahl der Loops
FEA6-   AD 59 FB    LDA   $FB59        ; Nummer des „alten" Units
FEA9-   29 70       AND   #$70
FEAB-   AA          TAX                ; = Slotno * 16, 00 für allerersten Aufruf
FEAC-   F0 0F       BEQ   $FEBD          der RWTB => Exit
FEAE-   20 DC FC    JSR   $FCDC        ; IS MOTOR ON? für „altes" Unit
FEB1-   F0 0A       BEQ   $FEBD        ; Motor ist aus: RTS
FEB3-   A9 01       LDA   #$01
FEB5-   20 85 FB    JSR   $FB85        ; sonst muß gewartet werden, bis das Mono-
FEB8-   AD 70 FB    LDA   $FB70          flop auf der „alten" Card abgelaufen ist
FEBB-   D0 E9       BNE   $FEA6        ; Loop für max. 255 Tests (reicht aus)
FEBD-   60          RTS
```

```
FEBE..FEFF      unbenutzt (Nullen)

FF00..FF9A      Interface zur RAMDisk-RWTB

FF9B..FFCA      IRQ Handler

FFCB..FFD7      RESET Handler

FFD8..FFEB      IRQ-Exit

FFEC..FFF9      unbenutzt (Nullen)

FFFA-   FB  03                          ;  NMI - Vektor
FFFC-   CB  FF                          ;  RESET - Vektor
FFFE-   9B  FF                          ;  IRQ - Vektor
```

# 5.7. Die RAMDisk

Die RAMDisk unter ProDOS setzt das Vorhandensein einer erweiterten 80-Zeichen Karte („Extended 80-Col Card") voraus. Eine derartige Karte ist (vorläufig) nur für die Maschinen IIe und IIc erhältlich. Es bleibt abzuwarten, ob eine entsprechende Erweiterung für den II+ von einem (Fremd)hersteller auf den Markt kommt. Dem steht allerdings entgegen, daß eine derartige Karte für den II+ reichlich kompliziert ausfallen dürfte, wenn dabei die Originalkarte wirklich vollständig emuliert werden soll.

Aufgrund der Tatsache, daß das „Technical Reference Manual" für den IIe, in dem das Bank-Switching der erweiterten 80-Col Card recht ausführlich erklärt ist, zum Zeitpunkt der Erstellung dieses Buches für mehrere Monate nicht erhältlich war (und trotz diverser Beteuerungen immer noch nicht erhältlich ist), folgt hier eine etwas ausführlichere Erklärung der Funktionsweise dieser Karte.

Die Karte ermöglicht die Darstellung von 80 Zeichen („80-Col Card") und hat darüberhinaus zusätzlich 64k RAM („Extended 80-Col Card").
Beim Betrieb mit 80 Zeichen wird der „normale" Bildspeicher des APPLE ($0400..$07FF) sowie der Speicherbereich $0400..$07FF innerhalb der zusätzlichen 64k benutzt. Da für den Prozessor softwaremäßig kein Unterschied zwischen dem „normalen" Speicherbereich („MAIN MEMORY") und dem zusätzlichen Speicherbereich („AUX MEMORY") besteht, wird vor der Schreibaktion (oder Leseaktion durch z.B. Rechtspfeil-Taste) die entsprechende Speicherbank selektiert.
Das Programm, das die Bank-Umschaltungen und die Cursorpositionierung erledigt, ist bereits innerhalb der ROM's auf der Hauptplatine des APPLE, nicht in einem ROM auf der Karte selber, was von den Adressen her auf dasselbe hinausläuft: Die entsprechenden Routinen finden sich beim Apple IIe im Bereich $C300..$C3FF sowie auf $C800..$CFFF

(EXROM) nach Ansprechen einer Adresse im Bereich $C300..$C3FF. Die Karte verhält sich also wie eine normale Zusatzkarte in Slot 3.
(Anmerkung: Der Apple IIc verfügt über völlig umgeschriebene ROM's durch die Integration der Mouse-Software, der seriellen Schnittstelle und des 65C02-Prozessors und den Wegfall der Slots. Viele Einsprungadressen sind erhalten geblieben (nicht alle!), auch wenn die ausführenden Routinen teils völlig woanders liegen. So wanderte z.B. MOVE von $C363 (IIe) nach $CF86 (IIc), der Einsprung ist immer noch bei $C311.)

Bei einer Karte, die nicht „Extended" ist, d.h. nur die 80-Zeichen Darstellung erlaubt, aber nicht über zusätzliche 64k RAM verfügt, ist nur der Bereich $0400..$07FF mit RAM auf der Karte selber belegt (1 kByte).

Um bei einer „extended"-Karte den Datentransport von normalen 64k („MAIN") zu den zusätzlichen 64k („AUX") zu ermöglichen, können verschiedene Speicherbereiche geschaltet werden:

a) **ALTZP:** Bereich $0000..$01FF und $D000..$FFFF
$C008 – MAIN Zeropage/Stack sowie ROM oder Language Card
$C009 – AUX Zeropage und Stack sowie ROM oder Bereich $D000..$FFFF in AUX
$C016 – Read ALTZP Switch (Bit 7 gesetzt, wenn AUX selektiert)
Durch das Setzen dieses Switches wird sowohl READ als auch WRITE gesetzt, d.h. es ist nicht möglich, von MAIN zu lesen und gleichzeitig auf AUX zu schreiben.
Die Schaltung der oberen 16k ($D000..$FFFF) erfolgt auch innerhalb des AUX-Bereichs mit den Speicherstellen für die Language Card, also $C080..$C08F – nur daß hier zwischen ROM und den Banks von AUX geschaltet wird!

b) **RAMRD/RAMWRT:** Bereich $0200..$BFFF
$C002 – Read MAIN
$C003 – Read AUX
$C004 – Write MAIN
$C005 – Write AUX
$C013 – Read RAMRD Switch ($C002/03) (Bit 7 gesetzt, wenn Read von AUX)
$C014 – Read RAMWRT Switch ($C004/05) (Bit 7 gesetzt, wenn Write nach AUX)
Mit diesen beiden Schaltern wird für den Bereich $0200..$BFFF bestimmt, aus welcher Bank (MAIN oder AUX) der Prozessor liest (also ggf. auch das eigene Programm!) und in welche Bank die Schreibaktionen stattfinden. Hier ist es möglich, von einer Bank direkt in die andere zu kopieren, in dem man z.B. „Read von MAIN" und „Write nach AUX" setzt und einen Loop aufbaut, der eigentlich widersinnig aussieht:

```
    LDY #$00
L1: LDA $2000,Y
    STA $2000,Y
    INY
    BNE L1
```

Anmerkung: Selbstmodifizierender Code zeitigt dabei erstaunliche Ergebnisse: Eine Modifikation des Loop mit „INC L1+2", also dem Erhöhen der Quelladresse, schreibt dann auf die Ziel-Bank und nicht in den Programmcode!

c) Ein dritter Schalter wird innerhalb der RAMDisk-RWTB noch benutzt:
$C000 – 80STORE aus – Video-Display immer von MAIN
$C001 – 80STORE an – Video-Display von selektierter Bank (MAIN oder AUX)
$C018 – Read 80STORE Switch (Bit 7 gesetzt, wenn 80STORE an)
Innerhalb der RWTB wird dieser Schalter auf OFF gesetzt, um ein Flackern des Video-Displays während der RAMDisk-Zugriffe zu vermeiden. Vor dem Exit der RWTB wird der ursprüngliche Status wieder hergestellt.

Die RAMDisk hat ein paar Besonderheiten:

a) Die Volume Directory umfaßt nur einen Block. Damit sind maximal 12 File-Einträge möglich. Subdirectories können dagegen beliebig lang sein.

b) Das Kommando FORMAT ($03) ist implementiert.

c) Der Bereich $0C00..$0DFF ist als „Puffer-Block" reserviert. Über diesen Bereich wird der Zugriff auf die VBM und auf Blocks im LC-Bereich des AUX-RAM ($D000..$FFFF) abgewickelt.

d) Die Blocks 00, 01, 04, 05, 06 lassen sich nicht beschreiben, READ liefert immer einen leeren Block zurück.

e) Block 07 liefert bei einem READ Zeropage und Stack von MAIN zurück, bei einem WRITE wird dieser Bereich in AUX beschrieben. Da dürfte der Programmierer Murks gemacht haben.

f) Die VBM existiert nicht als eigener Block auf der RAMDisk, sondern ist innerhalb der AUX-RWTB gespeichert und wird bei Bedarf in den Puffer-Block kopiert und von da aus gelesen (READ) bzw. in den Puffer-Block geschrieben und von da aus in die AUX-RWTB kopiert (WRITE).

g) Nach der Installation ist die RAMDisk „unformatiert". Das erste COMMAND (außer $00) führt zu einem internen FORMAT, danach wird das COMMAND selber bearbeitet. Zu diesem Zweck hat die AUX-RWTB ein INITFLAG, das nach einem FORMAT (intern oder explizit) gesetzt wird. Leider wird dieses Flag (zusammen mit der gesamten AUX-RWTB) bei einem „-PRODOS"-Kommando neu geschrieben. Deshalb geht bei jedem Neustart von PRODOS der Inhalt der RAMDisk verloren.
(Bei einem Kaltstart des BASIC.SYSTEM bleibt die RAMDisk erhalten).

### 5.7.1. Der Aufbau der RAMDisk und die Speicherbelegung innerhalb der AUX 64k

| | |
|---|---|
| $0000..$01FF | Zeropage und Stack |
| $0200..$03FF | RAMDisk-RWTB |
| $0400..$07FF | Bildspeicher für 80-Col Betrieb |
| $0800..$0BFF | beim APPLE IIe nicht benutzt – beim IIc z.T. Zwischenspeicher für serielle Schnittstelle und Tastatur |

| | |
|---|---|
| $0C00..$0DFF | „Puffer-Block" für Transfer LC-Bereich und VBM („Block 03") |
| $0E00..$0FFF | Volume Directory („Block 02") |
| $1000..$19FF | Blocks $09,$1A,$2B,$3C,$4D:<br>Block $09 ($1000), Block $1A ($1200), Block $2B ($1400),<br>Block $3C ($1600), Block $4D ($1800) |
| $1A00..$1FFF | Blocks $5D,$5E,$5F:<br>Block $5D ($1A00), Block $5E ($1C00), Block $5F ($1E00); |
| $2000..$3FFF | Block $08 ($2000), Block $0A ($2200) ... Block $18 ($3E00) |
| $4000..$5FFF | Block $19 ($4000), Block $1B ($4200) ... Block $29 ($5E00) |
| $6000..$7FFF | Block $2A ($6000), Block $2C ($6200) ... Block $3A ($7E00) |
| $8000..$9FFF | Block $3B ($8000), Block $3D ($8200) ... Block $4B ($9E00) |
| $A000..$BFFF | Block $4C ($A000), Block $4E ($A200) ... Block $5C ($BE00) |
| | **Bank 1** des LC-Bereichs in AUX: Blocks $60..$67 |
| $D000..$DFFF | Block $60 ($D000), Block $61 ($D200), Block $62 ($D400) .... |
| | **Bank 2** des LC-Bereichs in AUX: Blocks $68..$6F |
| $D000..$DFFF | Block $68 ($D000), Block $69 ($D200), Block $6A ($D400) .... |
| $E000..$FFFF | Blocks $70..$7F<br>Block $70 ($E000), Block $71 ($E200), Block $72 ($E400) .... |

Innerhalb des LC-Bereichs ist die Blockbelegung konsekutiv, d.h. aufeinanderfolgende Blocknummern belegen aufeinanderfolgende Speicherplätze im RAM. Warum die Speicherbelegung innerhalb der unteren 48k des AUX-RAM so überaus kompliziert ausgefallen ist, läßt sich nicht ergründen.

## 5.7.2. Blockbeschreibung des Interfaces der RAMDisk-RWTB

Im MLI werden die RWTB-Parameter $42..$47 (COMMAND, Unitnummer, Ziel/Quelladresse, Blocknummer) gesetzt, via DO DISK I/O wird $FF00 aufgerufen.

| | |
|---|---|
| $FF00 | Die Speicherstellen $003C..$0047 werden zwischengespeichert, genauso wie die Speicherstellen $03ED/EE ($003C..$0043 werden innerhalb der RWTB als Pointer benutzt, $03ED/EE erhalten die „Returnadresse" von XFER, der Bankswitch-Routine). |
| $FF16 | Wenn das COMMAND = $00 ist, folgt sofort ein Sprung zu EXIT ($FF44), die RAMDisk ist immer „Ready" und nie WRITE PROTECTED. |
| $FF1A | Das COMMAND muß im Bereich $00..$03 sein, sonst folgt I/O ERROR. |

$FF1E Aus COMMAND XOR $03 ergibt sich: FORMAT($03) = $00, WRITE($02) = $01 und READ($01) = $02. Bei FORMAT wird der Test auf „maximale Blocknummer" übersprungen.

$FF24 Test auf maximale Blocknummer, bei Blockno > $7F folgt I/O ERROR.

$FF2C $03ED/EE erhalten als „Returnadresse" $0200.

$FF33 Der Sprung zu XFER mit SEC/CLV schaltet den Bereich $0200..$BFFF auf „Read von AUX, Write nach AUX" und springt zur „Returnadresse" in AUX. Der restliche Speicherbereich (Zeropage/Stack und LC-Bereich) bleibt auf MAIN geschaltet.

$FF3B **Error Exits**
ERR#27: I/O ERROR. Angesprungen bei Überschreitung der maximalen Blocknummer ($7F) oder einem COMMAND außerhalb des Bereichs $00-$03. Weiter mit $FF47.
ERR#2B: WRITE PROTECTED. Von nirgendwoher angesprungen.

$FF44 Command ausgeführt, ERR#00 (kein Fehler).
Hierher wird entweder von $FF22 („$00"-Command) oder vom Ende der RWTB in AUX via XFER gesprungen.

$FF47 **EXIT**
Restore von $003C..$0047 auf der (MAIN) Zeropage sowie von $03ED/EE. Sämtliche Speicherbereiche sind (wieder) auf MAIN geschaltet, die Bank 1 der Language Card ist aktiviert (+ WRITE ENABLE).
RTS mit CLC und $00 im Akku, wenn kein Fehler, ansonsten mit SEC und ERR# im Akku zu der Routine, die DO DISK I/O innerhalb des MLI aufgerufen hat.

$FF62..$FF7E **WRITE Transfer Loop**
Dieser Loop wird von der RWTB in AUX via XFER aufgerufen, wenn von MAIN nach AUX geschrieben werden soll (WRITE auf die RAMDisk).
Die Pointer zur Zieladresse (entweder direkte Block-Adresse oder die Puffer-Adresse $0C00/$0D00) sowie die Quelladressen-Pointer sind innerhalb der AUX-RWTB gesetzt worden.
Es werden $200 Byte von MAIN nach AUX kopiert, danach Sprung zu $02D8 in AUX via XFER (von $FF33 her angesprungen).

$FF7F/80 Zwischenspeicher für $03ED/EE. Beim Exit zurückgeladen.

$FF81..$FF8C Zwischenspeicher für $003C..$0047. Beim Exit zurückgeladen.

## 5.7.3. Die RAMDisk-RWTB innerhalb der zusätzlichen 64k (AUX)

$0200..$0212 Sprung hierher nur von $FF2C via XFER, der Bereich $0200..$BFFF ist auf AUX geschaltet (Read und Write), Zeropage/Stack und LC-Bereich sind auf MAIN gesetzt.

Der Status von 80STORE wird gespeichert, 80STORE wird abgeschaltet, d.h. das Video-Display liest ab jetzt immer von MAIN.
Die RWTB-Parameter werden auf $03BB..$03BF (in AUX hinein) kopiert.

$0214 Wenn das COMMAND nicht FORMAT ist und die RAMDisk bereits „formatiert" wurde, Sprung zu $024D.

$0219..$024C **FORMAT**
Für den Fall, daß die RAMDisk noch nicht „formatiert" ist und das COMMAND nicht FORMAT war, wird die Blocknummer ($46) im X-Register zwischengespeichert.
Der Bereich $0E00..$0FFF (Volume Directory Block) wird via $0331 gelöscht, von $03D0 wird der erste Teil des Headers (Key Block-Eintrag) kopiert.

$022B Die „VBM" wird auf $03C0..$03CF angelegt, Block $00 bis $07 sind bereits belegt, die Blocks $08..$7F sind frei.

$0237 Der zweite Teil des Volume Directory Header (ACCESS, Eintragslänge..) wird von $03D4 in den Directory Block kopiert.

$0242 Wenn die RAMDisk bereits „formatiert" war, dann war das COMMAND tatsächlich FORMAT: Exit via $02A8 zu $FF44 in MAIN. Ansonsten wird jetzt die Blocknummer, die die ganze Zeit im X-Register unverändert geblieben ist, gesetzt.

$024D **READ/WRITE**
Aus der Blocknummer * 2 ergibt sich eine Bereichseingrenzung der Block-Adresse innerhalb der RAMDisk.
Wenn die Block-Adresse > $BFFF ist, Sprung zu $0261.
Wurde Block $03 (VBM) adressiert, Sprung zu $038A, ansonsten Sprung zu $0340.

$0261..$02BB Die Block-Adresse liegt im Bereich $C000..$FFFF.
Der erste Aufruf von $02E3 ist mal wieder eine der berüchtigten Speicherplatzeinsparungen des ProDOS-Programmierers: Die Pointer für den Block-Transport werden (falsch) gesetzt, wichtig ist hier nur das Carry (gesetzt bei WRITE, gelöscht bei READ).

$0265 Das Carry wird gepusht. Wenn das COMMAND WRITE war, wird via $02B6 der Puffer-Block als Zieladresse gesetzt und von MAIN (Quelle) nach AUX ($0C00/$0D00) kopiert.

$0268 Wenn die Block-Adresse im Bereich $C000..$CFFF liegt (Blocks $60..$67), wird sie um $1000 erhöht, die Bank 1 bleibt selektiert.
Ansonsten wird die Bank 2 des LC-Bereichs angeschaltet. Momentan ist immer noch MAIN für diesen Bereich selektiert!

$027A Die Quell/Zieladresse in MAIN wird auf den Stack gebracht, danach wird ALTZP geschaltet, d.h. Zeropage/Stack und der LC-Bereich sind jetzt ebenfalls auf AUX gesetzt. Der Pufferbereich $0C00/$0D00 wird als Quell/Zieladresse gesetzt, der Aufruf von $02E3 setzt die Pointer auf

der (AUX) Zeropage so, daß der auf $0292 folgende Loop immer in die „richtige Richtung" kopiert, d.h. bei READ wird vom LC-Bereich nach $0C00/$0D00 kopiert, bei einem WRITE wird von $0C00/$0D00 auf den Ziel-Block geschrieben.

$029D Zurückschaltung von Zeropage/Stack und LC-Bereich auf MAIN, die Zieladresse wird wieder vom Stack geholt, für WRITE ist das die Adresse des Ziel-Blocks in AUX, sie wird nicht mehr gebraucht.

$02A7 Das Carry von der Unterscheidung READ/WRITE (s. $0265) wird vom Stack geholt, für WRITE (Carry gesetzt) folgt Exit zu $FF44 via XFER, vorher wird noch für alle Fälle wieder die Bank 1 der LC angeschaltet.
Für READ muß hier auf Bank 1 der LC zurückgeschaltet werden, danach folgt ein Aufruf des „regulären" Block-Transports, der jetzt von $0C00/$0D00 in AUX auf die Zieladresse in MAIN kopiert. Danach ebenfalls Exit via XFER zu $FF44.

$02B9 Der „Patch" für WRITE: Aufruf von $02BC zum Kopieren von MAIN (Quelle) nach $0C00/$0D00 in AUX, danach Sprung zu $0268 (s. $0266).

$02BC..$02E3 Setzen der Ziel/Quellpointer und Blocktransport zwischen MAIN und AUX

$02BC Einsprung für das Setzen der Quell/Zieladresse in AUX auf den Puffer-Block $0C00/$0D00.

$02BE Einsprung mit AUX-Quell/Zieladresse im Akku nach der Berechnung der Block-Adresse in AUX.

$02C1 Einsprung für READ/WRITE der VBM, Aufruf von $02E3 zum Setzen der Quell/Zielpointer für MAIN und AUX. RTS von $02E3 mit Carry gesetzt bei WRITE, ansonsten mit Carry gelöscht.

$02C4 Wenn das COMMAND WRITE war, wird via XFER MAIN im Bereich $0200..$BFFF selektiert und zu $FF62 im MLI (MAIN) gesprungen. Die Routine auf $FF62 setzt „Write nach AUX", kopiert $200 Byte von MAIN (Quelle) nach AUX (Ziel-Block entweder im Bereich $0200..$BFFF oder Puffer-Block $0C00/$0D00), und springt danach (wieder via XFER) zu $02D8 in AUX. Hier steht ein RTS, der zu der Routine führt, die $02BC ($02BE, $02C1) aufgerufen hat.
Wenn das COMMAND READ war, wird „Write nach MAIN" gesetzt und von AUX (Quell-Block oder Puffer-Block $0C00/$0D00) nach MAIN (Zielbereich) kopiert. Dabei kann der Zielbereich in MAIN auch in der (sowieso angeschalteten) Language Card liegen. Danach folgt Rückschaltung auf „Write auf AUX" und RTS zu der Routine, die $02BC ($02BE, $02C1) aufgerufen hat.

$02E3..$0325 Setzen der Pointer für den Blocktransport und des Carry-Flags (gelöscht = READ, gesetzt = WRITE).
Für READ werden $003C..3F auf den Quell-Block (oder $0C00/$0D00) innerhalb von AUX und $0040..43 auf die Zieladresse innerhalb von MAIN gesetzt (Ausnahme: READ von einem Block im AUX-Bereich $D000..$FFFF, hier wird beim ersten Durchlauf die Zieladresse $0C00/$0D00 gesetzt).

Für WRITE werden $003C..3F auf den Ziel-Block in AUX (tatsächliche Adresse oder $0C00/$0D00) gesetzt, $0040..43 auf die Quell-Adresse innerhalb von MAIN (Ausnahme: WRITE auf einen Block im AUX-Bereich $D000..$FFFF, hier wird beim zweiten Durchlauf die Quelladresse $0C00/$0D00 innerhalb von AUX gesetzt).

$0326..$032E Handler für die Blocks 0,1,4,5,6 – die Blocks 2 und 3 (VolDIR und VBM) werden vorher „abgefangen".
Via $032F wird der Puffer-Block in AUX gelöscht, danach folgt ein „normaler" Transfer (READ oder WRITE, je nach COMMAND) – nur daß die Adresse in AUX (Zieladresse für WRITE, Quell-Adresse für READ) immer $0C00 ist, d.h. es passiert gar nichts. Man kann diese „Blocks" zwar schreiben, aber für das Lesen ergibt sich immer ein leerer Block.
Exit via XFER zu $FF44 im MLI.

$032F..$033F Löschen eines Blocks in AUX
$032F Einsprung zum Löschen des Puffer-Blocks $0C00/$0D00.
$0331 Einsprung für FORMAT. Hier ist die Zieladresse $0E00, d.h. der Volume Directory-Block wird gelöscht.
Via $02E3 werden die Pointer gesetzt, danach wird der „Quell-Bereich" gelöscht. RTS zur aufrufenden Routine.

$0340..$0359 Handler für die Blocks $00..$5F (Blockadresse im Bereich $0000..$BFFF)
Der Block $03 (VBM) ist bereits vorher „abgefangen" worden.
Sprung hierher von $025E mit Blocknummer * 2 im Akku.
Wenn die Volume Directory (Block $02) adressiert werden soll, wird die Block-Adresse $0E00 (7 * 2) zugewiesen (Sprung zu $0383).
Wenn die Blocknummer größer als $5C ist, werden $50 abgezogen, danach Sprung zu $0383. Dadurch ergeben sich die Startadressen der Blocks $5D, $5E und $5F mit $1A00/$1C00/$1E00.
Die Blocknummer wird auf eine entsetzlich komplizierte Weise in die Block-Startadresse innerhalb von AUX umgerechnet:
RES = Blocknummer = Blocknummer - 8
RES = INT (RES / $11) + Rest (Y-Reg)
IF Rest = 1 THEN RES = RES + 08: ergibt Blocks $09, $1A, $2B, $3C, $4D
ELSE RES = RES * $10
IF Rest = 0 THEN RES = RES + Rest: ergibt Blocks $08, $19, $2A, $3B, $4C
ELSE Rest = Rest-1 : RES = RES + Rest :ergibt das „Zusammenschieben" der restlichen Blocks und damit die lückenlose Reihenfolge der Blocks $08, $0A, $0B / $19, $1B, $1C...

Die Blocknummer $07 übersteht diesen Prozeß nicht so ganz, hier ist das Ergebnis $00. Die Folge ist ein „READ" der MAIN-Zeropage und des

MAIN-Stacks bei einer Adressierung dieser Blocknummer via READ BLOCK, WRITE BLOCK überschreibt Stack und Zeropage in AUX, was auch höchstens für „Kopierschützer" interessant wäre.

$0383 Die Block-Startadresse wird mal 2 genommen, via $02BE werden die Pointer gesetzt und der Block-Transfer von MAIN nach AUX und umgekehrt erledigt. Exit via XFER zu $FF44.

$0384..$03B9 READ/WRITE des VBM-Blocks
Der Puffer-Block $0C00/$0D00 wird als Quell/Ziel-Block in AUX gesetzt, via $02E3 werden die Pointer gesetzt.
Wenn das COMMAND READ war (Carry gelöscht nach $02E3), wird der Puffer-Block via $0334 gelöscht und die VBM von $03C0..$03CF hineinkopiert. Danach folgt ein „regulärer" Transport auf die Zieladresse in MAIN via $02C1 und Exit via XFER zu $FF44 in MAIN.
Wenn das COMMAND WRITE war, wird via $02C1 von MAIN (Quelle) auf den Puffer-Block $0C00/$0D00 in AUX kopiert. Danach werden die Pointer noch einmal gesetzt (Ziel-Pointer $0042/43 auf $0C00), die ersten $10 Bytes der VBM werden von $0C00..$0C0F in den Bereich $03C0..$03CF kopiert. Exit via XFER zu $FF42 in MAIN.

$03BA **INITFLAG:** $FF = „bereits formatiert", $00 bewirkt, daß beim ersten COMMAND die RAMDisk „formatiert" wird, d.h. die Volume Directory (nur Header) und die VBM ($03C0..$03CF) werden erstellt.

$03BB..$03BF Zwischenspeicher für die RWTB-Parameter.

$03C0..$03CF **VBM-Bereich der RAMDisk.** Wird bei READ der VBM (Block $03) in den Puffer-Block $0C00/$0D00 kopiert und von da aus nach MAIN geschrieben, bei WRITE wird zuerst von MAIN nach $0C00/$0D00 gelesen und dann hierher kopiert.

$03D0..$03DB Volume Directory Header-Informationen, werden bei FORMAT in den Header der VolDIR ($0E04..$0E29) kopiert.

$03DC..$03FD **EXIT der AUX-RWTB**
Bank 1 der Language Card wird selektiert (für den Fall, daß innerhalb der RWTB die Bank 2 von AUX angesprochen wurde), der Status von 80STORE wird vom (MAIN) Stack geholt und 80STORE wird wieder entsprechend gesetzt. $FF44 in MAIN wird als „Returnadresse" gesetzt.

$03F6 CLC/CLV: $0200..$BFFF von MAIN, Zeropage/Stack und LC-Bereich von MAIN, Sprung zu XFER.

## 5.7.4. Kommentierte Listings der RAMDisk-RWTB

### A. Der Test auf „Extended 80-Col Card" im Relocator

Da hier zum Test der Bereich $0200..$BFFF umgeschaltet wird, muß die Prüfroutine selber außerhalb dieses Bereichs liegen und wird deshalb nach $0080 kopiert.
Um zwischen „extended" und „normaler" 80-Zeichen Karte zu unterscheiden, wird ein Bereich getestet, der außerhalb des zusätzlichen Bildspeichers ($0400..$07FF) liegt.

– Test auf „128k"; diese Routine wird nach $0080 kopiert

```
2401-   85 0C       STA   $0C          ; MACHID
2403-   10 35       BPL   $243A        ; nicht IIe/IIc: kein Test auf „128k"
2405-   A9 EE       LDA   #$EE
2407-   8D 05 C0    STA   $C005        ; WRITE nach AUX
240A-   8D 03 C0    STA   $C003        ; READ von AUX
240D-   8D 00 0C    STA   $0C00
2410-   8D 00 08    STA   $0800        ; diese Schreib- und Leseaktionen finden
2413-   AD 00 0C    LDA   $0C00          jetzt innerhalb der zusätzlichen
2416-   C9 EE       CMP   #$EE           64k statt!
2418-   D0 0E       BNE   $2428        ; keine extended 80-Col Card
241A-   0E 00 0C    ASL   $0C00
241D-   0A          ASL
241E-   CD 00 0C    CMP   $0C00
2421-   D0 05       BNE   $2428        ; nein
2423-   CD 00 08    CMP   $0800        ; na ja... ($800 darf sich nicht ändern)
2426-   D0 03       BNE   $242B        ; ok. hat 128k
2428-   38          SEC
2429-   B0 01       BCS   $242C
242B-   18          CLC
242C-   8D 04 C0    STA   $C004        ; Zurückschalten: Write nach MAIN
242F-   8D 02 C0    STA   $C002        ; READ von MAIN
2432-   B0 06       BCS   $243A        ; C gesetzt, wenn keine RAMDisk
2434-   A5 0C       LDA   $0C
2436-   09 30       ORA   #$30         ; 128k (64k RAMDisk) – Setzen von Bit 4/5
2438-   85 0C       STA   $0C
243A-   ......
```

### B. Installation der RAMDisk-RWTB

Die RAMDisk-RWTB ist auf mehrere Speicherbereiche (und Banks) verteilt:
a) Das „Interface" innerhalb des MLI ($F000..$FF61 und $FF7F..$FF8C);
b) Die Block-Kopierroutine von MAIN nach AUX innerhalb des MLI ($FF62..$FF7E);
b) Der größte Teil der RWTB selber in AUX auf $0200..$03FF;
c) Das Bank-Switching zwischen MAIN und AUX auf $C300..$C3FF („Slot 3" IIe).

– Test von MACHID im Relocator nach Installation von REBOOT

```
20C8-   AD 98 BF    LDA   $BF98        ; MACHID
20CB-   29 30       AND   #$30
20CD-   49 30       EOR   #$30
20CF-   D0 03       BNE   $20D4        ; wenn Bit 4/5 gelöscht => keine „128k"
20D1-   20 FF 28    JSR   $28FF
20D4-   ....
```

– Installation der RAMDisk-RWTB

```
28FF-   EA          NOP
2900-   A0 99       LDY   #$99
2902-   B9 00 2C    LDA   $2C00,Y
2905-   99 00 FF    STA   $FF00,Y       ; das Interface zur RAMDisk-RWTB wird
2908-   88          DEY                   von $2C00 nach $FF00..$FF99 in der
2909-   C0 FF       CPY   #$FF            Language Card kopiert
290B-   D0 F5       BNE   $2902

290D-   A2 00       LDX   #$00
290F-   86 3C       STX   $3C
2911-   CA          DEX
2912-   86 3E       STX   $3E
2914-   A2 2A       LDX   #$2A          ; Setup für AUXMOVE
2916-   86 3D       STX   $3D
2918-   E8          INX                 ($3C): $2A00 (Quell-Startadresse)
2919-   86 3F       STX   $3F
291B-   A9 00       LDA   #$00          ($3E): $2BFF (Quell-Endadresse)
291D-   85 42       STA   $42
291F-   A9 02       LDA   #$02          ($42): $0200 (Zieladresse)
2921-   85 43       STA   $43
2923-   38          SEC                 ; SEC: Read von MAIN, Write nach AUX
2924-   20 11 C3    JSR   $C311         ; kopiert $2A00..$2BFF auf AUX $0200..$03FF

2927-   A9 00       LDA   #$00
2929-   8D 26 BF    STA   $BF26         ; der Start der RAMDisk-RWTB ($FF00)
292C-   A9 FF       LDA   #$FF            wird in die DRIVER TABLE eingetragen
292E-   8D 27 BF    STA   $BF27
2931-   EE 31 BF    INC   $BF31         ; DEVNUM wird um eins erhöht
2934-   AE 31 BF    LDX   $BF31
2937-   A9 BF       LDA   #$BF          ; RAMDisk als „Slot 3, Drive 2"
2939-   9D 32 BF    STA   $BF32,X         in die DEVICE TABLE
293C-   60          RTS
```

Die Routinen AUXMOVE und XFER im Bereich $C300..$C3FF. Der Vollständigkeit halber ist hier auch noch der „PR#3"-Einsprung mit gelistet:
(Anmerkung: Nur gültig für IIe, für den IIc stimmen nur die Einsprungadressen, die ausführenden Routinen stehen im Bereich $C800..).

```
C300-   2C 58 FF    BIT   $FF58         ; Set V-Flag: Einsprung für „PR#3"
C303-   70 12       BVS   $C317         ; „always"
C305-   38          SEC                 ; „Key in" – (KSW-Vektor)
C306-   ** 90 **                        ; von $C305 her ist das ein BCC „never"
C307-   18          CLC                 ; „Char out" – (CSW-Vektor) – CLC von
C308-   B8          CLV                   $C305 her maskiert
C309-   50 0C       BVC   $C317         ; „always"

C30B-   01 88 4B 51 57 5D               ; $C305/$C307/$C30B/$C30C = ID der Karte

C311-   4C 63 C3    JMP   $C363         ; AUXMOVE (IIc JMP $CF86)

C314-   4C B0 C3    JMP   $C3B0         ; XFER (IIc JMP $CFCD)

C317..C362      Routinen für 80 Col-Betrieb
```

**– AUXMOVE**

Diese Routine kopiert entweder von MAIN nach AUX oder umgekehrt. Ein Kopieren innerhalb einer der beiden Speicherbanks ist nicht möglich.

Aufruf mit Carry gesetzt: MAIN => AUX
Aufruf mit Carry gelöscht: AUX => MAIN

Die Ziel- und Quellpointer müssen vorher gesetzt sein:

$003C/3D: Quell-Start, wird während der Kopie verändert (erhöht)
$003E/3F: Quell-Ende, bleibt unverändert
$0042/43: Ziel-Start, wird während der Kopie verändert (erhöht)
(Anmerkung: Beim IIc beginnt MOVEAUX auf $CF86, die Funktion ist dieselbe.)

```
C363-   48          PHA
C364-   98          TYA             ; Save von A-Y
C365-   48          PHA
C366-   AD 13 C0    LDA   $C013     ; Read RAMRD Switch
C369-   48          PHA             ; Save des vorherigen Status
C36A-   AD 14 C0    LDA   $C014     ; Read RAMWRT Switch
C36D-   48          PHA

C36E-   90 08       BCC   $C378
C370-   8D 02 C0    STA   $C002     ; Read von MAIN
C373-   8D 05 C0    STA   $C005     ; Write nach AUX
C376-   B0 06       BCS   $C37E
C378-   8D 04 C0    STA   $C004     ; Write nach MAIN
C37B-   8D 03 C0    STA   $C003     ; Read von AUX

C37E-   A0 00       LDY   #$00
C380-   B1 3C       LDA   ($3C),Y   ; Kopie von $003C/3D: Quelle
C382-   91 42       STA   ($42),Y   ; nach $0042/43: Ziel
C384-   E6 42       INC   $42
C386-   D0 02       BNE   $C38A
C388-   E6 43       INC   $43
C38A-   A5 3C       LDA   $3C
C38C-   C5 3E       CMP   $3E       ; Vergleich mit Quell-Obergrenze: $003E/3F
C38E-   A5 3D       LDA   $3D
C390-   E5 3F       SBC   $3F
C392-   E6 3C       INC   $3C
C394-   D0 02       BNE   $C398
C396-   E6 3D       INC   $3D
C398-   90 E6       BCC   $C380     ; Loop, bis Obergrenze erreicht

C39A-   8D 04 C0    STA   $C004     ; Write nach MAIN
C39D-   68          PLA
C39E-   10 03       BPL   $C3A3     ; Restore der vorherigen Konfiguration
C3A0-   8D 05 C0    STA   $C005     ; Write nach AUX

C3A3-   8D 02 C0    STA   $C002     ; Read von MAIN
C3A6-   68          PLA
C3A7-   10 03       BPL   $C3AC
C3A9-   8D 03 C0    STA   $C003     ; Read von AUX

C3AC-   68          PLA
C3AD-   A8          TAY             ; Restore von A-Y
C3AE-   68          PLA
C3AF-   60          RTS
```

– **XFER:** Bank-Switching zwischen MAIN und AUX

Aus Carry gelöscht/gesetzt und V-Flag gelöscht/gesetzt ergeben sich vier Möglichkeiten:

- Carry gesetzt: Read/Write AUX, Startadresse nach AUX
- Carry gelöscht: Read/Write MAIN, Startadresse nach MAIN
- V-Flag gesetzt: Zeropage/Stack von AUX und $D000..$FFFF von AUX oder ROM
- V-Flag gelöscht: Zeropage/Stack von MAIN und $D000..$FFFF von MAIN (LC) oder ROM

Die Umschaltung erfolgt in zwei Schritten: zuerst der Bereich $0200..$BFFF, danach Zeropage/Stack und LC-Bereich.
Da diese Routine auch den Stack umschalten kann, ist eine Fortsetzung des Programms via RTS nicht möglich und wird durch einen indirekten Sprung ersetzt.
(Anmerkung: Beim IIc beginnt XFER auf $CFCD, die Funktion ist dieselbe.)

```
C3B0-   48          PHA             ; Save der „Returnadresse":
C3B1-   AD ED 03    LDA   $03ED       der Speicherbereich $200..$BFFF wird
C3B4-   48          PHA               geschaltet, Zeropage/Stack und LC
C3B5-   AD EE 03    LDA   $03EE       dagegen (noch) nicht!
C3B8-   48          PHA
C3B9-   90 0A       BCC   $C3C5
C3BB-   8D 03 C0    STA   $C003     ; Carry gesetzt: Read von AUX
C3BE-   8D 05 C0    STA   $C005     ; und Write nach AUX
C3C1-   50 19       BVC   $C3DC
C3C3-   70 08       BVS   $C3CD
C3C5-   8D 02 C0    STA   $C002     ; Carry gelöscht: Read von MAIN
C3C8-   8D 04 C0    STA   $C004     ; und Write nach MAIN
C3CB-   50 0F       BVC   $C3DC
C3CD-   68          PLA             ; V-Flag gesetzt:
C3CE-   8D EE 03    STA   $03EE       Kopieren der „Returnadresse"
C3D1-   68          PLA               entweder nach AUX (C gesetzt) oder
C3D2-   8D ED 03    STA   $03ED       nach MAIN (C gelöscht)
C3D5-   68          PLA
C3D6-   8D 09 C0    STA   $C009     ; ALTZP (Zpage/Stack und LC von AUX)
C3D9-   6C ED 03    JMP   ($03ED)   ; „Return"

C3DC-   68          PLA             ; V-Flag gelöscht:
C3DD-   8D EE 03    STA   $03EE       Kopieren der „Returnadresse"
C3E0-   68          PLA               entweder nach AUX (C gesetzt) oder
C3E1-   8D ED 03    STA   $03ED       nach MAIN (C gelöscht)
C3E4-   68          PLA
C3E5-   8D 08 C0    STA   $C008     ; ALTZP (Zpage/Stack und LC von MAIN)
C3E8-   6C ED 03    JMP   ($03ED)   ; „Return"
```

## 5.7.5. Das Interface zur RAMDisk-RWTB innerhalb des MLI

```
FF00-   D8          CLD
FF01-   A2 0B       LDX   #$0B      ; die von dem RAMDisk-Routinen benutzten
FF03-   B5 3C       LDA   $3C,X       Speicherstellen auf der Zeropage
FF05-   9D 81 FF    STA   $FF81,X     ($003C..47) werden zwischengespeichert
FF08-   CA          DEX
FF09-   10 F8       BPL   $FF03

FF0B-   A2 01       LDX   #$01
FF0D-   BD ED 03    LDA   $03ED,X   ; $03ED/EE werden ebenfalls gespeichert, sie
FF10-   9D 7F FF    STA   $FF7F,X     werden innerhalb von XFER benutzt
FF13-   CA          DEX
FF14-   10 F7       BPL   $FF0D

FF16-   A5 42       LDA   $42       ; COMMAND
FF18-   F0 2A       BEQ   $FF44     ; bei „ACTIVATE" ($00) sofort Exit
FF1A-   C9 04       CMP   #$04      ; COMMAND > $04?
FF1C-   B0 1D       BCS   $FF3B     ; wenn ja, I/O ERROR
FF1E-   49 03       EOR   #$03      ; $01(READ)=>2, $02(WRITE)=>1, $03(FORMAT)=>0
FF20-   85 42       STA   $42
FF22-   F0 08       BEQ   $FF2C     ; FORMAT - kein Test auf max. Blocknummer
```

```
FF24-   A4 47       LDY   $47          ; Blocknummer hi
FF26-   D0 13       BNE   $FF3B        ; muß $00 sein, sonst I/O ERROR
FF28-   A5 46       LDA   $46          ; Blocknummer lo
FF2A-   30 0F       BMI   $FF3B        ; darf maximal $7F sein, sonst I/O ERROR

FF2C-   A9 00       LDA   #$00
FF2E-   8D ED 03    STA   $03ED
FF31-   A9 02       LDA   #$02         ; $03ED/EE => $0200 - Start der RWTB in
FF33-   8D EE 03    STA   $03EE          den zusätzlichen 64k (AUX)
FF36-   38          SEC                ; für XFER: Read und Write AUX
FF37-   B8          CLV                ; für XFER: Zeropage + Stack von MAIN
FF38-   4C 14 C3    JMP   $C314        ; XFER: schaltet um und springt zu $0200
                                         (innerhalb von AUX)
```

– Error Exit

```
FF3B-   A9 27       LDA   #$27         ; ERR#27: I/O ERROR
FF3D-   D0 02       BNE   $FF41
FF3F-   A9 2B       LDA   #$2B         ; ERR#2B: WRITE PROTECTED – nicht benutzt
FF41-   38          SEC
FF42-   B0 03       BCS   $FF47        ; „always"
```

– Exit ohne Fehler, von AUX aus via XFER angesprungen

```
FF44-   A9 00       LDA   #$00
FF46-   18          CLC
```

– Restore von $003C..$0047 und $03ED/EE

```
FF47-   08          PHP
FF48-   48          PHA
FF49-   A2 0B       LDX   #$0B
FF4B-   BD 81 FF    LDA   $FF81,X      ; Restore von $003C..47
FF4E-   95 3C       STA   $3C,X
FF50-   CA          DEX
FF51-   10 F8       BPL   $FF4B
FF53-   AD 7F FF    LDA   $FF7F
FF56-   8D ED 03    STA   $03ED        ; Restore von $03ED/EE
FF59-   AD 80 FF    LDA   $FF80
FF5C-   8D EE 03    STA   $03EE
FF5F-   28          PLP
FF60-   68          PLA
FF61-   60          RTS
```

– WRITE Transfer Loop, von AUX aus via XFER angesprungen

```
FF62-   8D 05 C0    STA   $C005        ; Write nach AUX
FF65-   A0 00       LDY   #$00
FF67-   B1 3C       LDA   ($3C),Y      ; Quell-Block, untere Hälfte
FF69-   91 42       STA   ($42),Y      ; Ziel-Block, untere Hälfte oder $0C00
FF6B-   B1 3E       LDA   ($3E),Y      ; Quell-Block, obere Hälfte
FF6D-   91 40       STA   ($40),Y      ; Ziel-Block, obere Hälfte oder $0D00
FF6F-   88          DEY
FF70-   D0 F5       BNE   $FF67        ; Loop für insgesamt $200 Bytes

FF72-   8D 04 C0    STA   $C004        ; Write nach MAIN
FF75-   A9 D8       LDA   #$D8         ; $02D8 in AUX: RTS, danach Sprung
FF77-   8D ED 03    STA   $03ED          zu $FF44 in MAIN via XFER
FF7A-   A9 02       LDA   #$02
FF7C-   4C 33 FF    JMP   $FF33        ; setzt $02D8, Sprung zu XFER
```

```
FF7F-   00 00                                           ; Zwischenspeicher für $03ED/EE
FF81-   00 00 00 00 00 00 00 00  00 00 00 00            ; Zwischenspeicher für $003C..47

FF8D-   00 00 00                                        ; nicht benutzt
FF90-   00 00 00 00 00 00 00 00  00 00
```

## 5.7.6. Die RAMDisk-RWTB innerhalb der zusätzlichen 64k (AUX)

Angesprungen von $FF2C via XFER. Zeropage und $D000..$FFFF sind auf MAIN, der Bereich $0200..$BFFF ist auf AUX (Read und Write) geschaltet. Die Language Card ist vom MLI auf Bank 1 gesetzt. Wenn ALTZP geschaltet wird, ist damit auch innerhalb von AUX die „LC-Bank 1" gesetzt.

```
0200-   AD 18 C0    LDA   $C018          ; Read 80STORE
0203-   48           PHA                  ; Save des Status von 80STORE
0204-   8D 00 C0    STA   $C000          ; Abschalten von 80STORE, d.h. das Display
0207-   EA          NOP                    bleibt unabhängig von der gewählten Bank
0208-   EA          NOP                    auf MAIN
0209-   EA          NOP
020A-   A2 04       LDX   #$04           ; $3BB: COMMAND (XOR $3), $3BC: Unitnum-
020C-   B5 42       LDA   $42,X            mer $3BD/BE: Quell/Zieladr., $3BF: Blockno
020E-   9D BB 03    STA   $03BB,X          Diese Werte werden von MAIN nach AUX
0211-   CA          DEX                    kopiert, also „hierher".
0212-   10 F8       BPL   $020C           Das letzte Byte ist $42 (COMMAND XOR 3)

0214-   2D BA 03    AND   $03BA          ; INITFLAG - schon „formatiert"?
0217-   D0 34       BNE   $024D          ; ja, COMMAND ist auch nicht 00 (FORMAT)
```

– **FORMAT:** entweder INITFLAG = $00 oder COMMAND XOR $03 = $00, d.h. COMMAND war $03 (FORMAT)

```
0219-   A6 46       LDX   $46
021B-   A9 0E       LDA   #$0E           ; Zieladresse für Löschen: $0E00/$0F00
021D-   20 31 03    JSR   $0331          ; löscht den Bereich $0E00..$0FFF (VolDIR)
0220-   A0 03       LDY   #$03
0222-   B9 D0 03    LDA   $03D0,Y        ; $F3 (Storage Type/Namenslänge) und „RAM"
0225-   99 04 0E    STA   $0E04,Y          wird in den Directory Header kopiert,
0228-   88          DEY                    Forward/Backward Pointer bleiben 00 00,
0229-   10 F7       BPL   $0222            die VolDIR hat nur einen Block.

022B-   98          TYA                  ; = $FF. Der VBM-Block wird konstruiert:
022C-   A0 0F       LDY   #$0F           ; $03C1..$03CF = $FF, $03C0 = $00, d.h. die
022E-   99 C0 03    STA   $03C0,Y          ersten 8 Blocks sind belegt,
0231-   88          DEY                    Blocks $08..$79 sind frei.
0232-   D0 FA       BNE   $022E
0234-   8C C0 03    STY   $03C0          ; = $00

0237-   A0 07       LDY   #$07           ; der Rest des VolDIR Header:
0239-   B9 D4 03    LDA   $03D4,Y          ACCESS = $C3, Eintragslänge: $27,
023C-   99 22 0E    STA   $0E22,Y          Anzahl der Einträge pro Block: $0D,
023F-   88          DEY                    aktive Einträge: 00 00, Blockno VBM: 00 03
0240-   10 F7       BPL   $0239           TOTAL BLOCKS: 00 80 ($0E2A = 00 v. Lösch.)

0242-   AD BA 03    LDA   $03BA          ; wenn INITFLAG bereits gesetzt, d.h.
0245-   D0 61       BNE   $02A8            COMMAND = FORMAT, dann Sprung
0247-   8C BA 03    STY   $03BA          ; sonst INITFLAG setzen ($FF) und Ausführung
024A-   8E BF 03    STX   $03BF            des COMMAND. X=Blockno (von $0219)
```

## – READ/WRITE COMMANDS

```
024D-  0E BF 03   ASL   $03BF        ; Blockno * 2
0250-  AD BF 03   LDA   $03BF
0253-  C9 BF      CMP   #$BF
0255-  B0 0A      BCS   $0261        ; Sprung, wenn Block-Adresse > $BFFF
0257-  C9 06      CMP   #$06         ; Block $03 (VBM)
0259-  D0 03      BNE   $025E
025B-  4C 8A 03   JMP   $038A        ; Spezialbehandlung für VBM-Block
025E-  4C 40 03   JMP   $0340        ; Transfer für Blocks im Bereich $0000..$BFFF
```

### – Blockadresse in AUX > $BFFF

```
0261-  AA         TAX            ** ; = Blockno * 2
0262-  20 E3 02   JSR   $02E3       ; nur Carry von „LSR COMMAND" wichtig!
0265-  08         PHP              ; C gelöscht bei READ
0266-  B0 4E      BCS   $02B6      ; WRITE: kopiert von MAIN (Quelle) nach
0268-  8A         TXA            ** $0C00/... (AUX) und springt wieder hierher
0269-  C9 CF      CMP   #$CF       ; Blockno * 2 > $CF?
026B-  B0 04      BCS   $0271      ; ja, immer BANKSELECT Bank 2
026D-  09 10      ORA   #$10       ; sonst Bereich $C000..$CFFF: => $D000..$DFFF
026F-  D0 06      BNE   $0277      ; „always"

0271-  8D 83 C0   STA   $C083      ; „LC-Bank 2" in AUX
0274-  8D 83 C0   STA   $C083      ; + WRITE ENABLE
0277-  8D BF 03   STA   $03BF      ; Block-Adr. hi für $C0..$CF Bank1 ($D0..$DF)
027A-  AD BE 03   LDA   $03BE      ; Quell/Zieladresse hi
027D-  48         PHA
027E-  AE BD 03   LDX   $03BD      ; Quell/Zieladresse lo
0281-  8D 09 C0   STA   $C009      ; ALTZP: $0..$01FF / $D000..$FFFF auf AUX
0284-  A9 0C      LDA   #$0C
0286-  8D BE 03   STA   $03BE      ; Quell/Zieladresse auf $0C00/$0D00
0289-  A9 00      LDA   #$00
028B-  8D BD 03   STA   $03BD
028E-  20 E3 02   JSR   $02E3      ; setzt die Transfer-Pointer
0291-  A8         TAY              ; immer = $00

0292-  B1 3C      LDA   ($3C),Y    ; Kopie innerhalb von AUX: bei READ wird
0294-  91 42      STA   ($42),Y      von der Blockadresse nach $0C00/$0D00,
0296-  B1 3E      LDA   ($3E),Y      bei WRITE von $0C00/$0D00 nach
0298-  91 40      STA   ($40),Y      Blockadresse kopiert!
029A-  88         DEY
029B-  D0 F5      BNE   $0292      ; Loop für $200 Byte
029D-  8D 08 C0   STA   $C008      ; ALTZP: $0000..$01FF / $D000..$FFFF auf MAIN
02A0-  8E BD 03   STX   $03BD      ; Restore der Quell/Zieladresse
02A3-  68         PLA
02A4-  8D BE 03   STA   $03BE      ; Quell/Zieladresse hi
02A7-  28         PLP              ; C gesetzt: WRITE (s. $0265)
02A8-  B0 09      BCS   $02B3      ; WRITE: BANKSELECT Bank 1 + Exit (=>$FF44)
02AA-  8D 8B C0   STA   $C08B      ; READ: BANKSELECT Bank 1
02AD-  8D 8B C0   STA   $C08B      ; + WRITE ENABLE
02B0-  20 BC 02   JSR   $02BC      ; kopiert jetzt von $0C00/... (AUX) nach MAIN
02B3-  4C DC 03   JMP   $03DC      ; Exit zu $FF44 via XFER

02B6-  20 BC 02   JSR   $02BC      ; WRITE: kopiert von MAIN nach $0C00/... (AUX)
02B9-  4C 68 02   JMP   $0268      ; weiter mit derselben Routine wie READ
```

### – Setzen der Pointer und Blocktransport MAIN <=> AUX

```
02BC-  A9 0C      LDA   #$0C       ; Adresse innerhalb AUX : Puffer $0C00/$0D00

02BE-  8D BF 03   STA   $03BF      ; Einsprung mit Block-Adresse im Akku
```

```
02C1-   20 E3 02    JSR   $02E3        ; Einsprung für VBM - Pointer werden gesetzt
02C4-   B0 13       BCS   $02D9        ; von „ASL COMMAND" ($02E3) Sprung f. WRITE
02C6-   8D 04 C0    STA   $C004        ; READ: Write nach MAIN
02C9-   A8          TAY
02CA-   B1 3C       LDA   ($3C),Y      ; und Kopie von $200 Byte auf die
02CC-   91 42       STA   ($42),Y        Zieladresse in MAIN
02CE-   B1 3E       LDA   ($3E),Y
02D0-   91 40       STA   ($40),Y
02D2-   88          DEY
02D3-   D0 F5       BNE   $02CA
02D5-   8D 05 C0    STA   $C005        ; Write nach AUX
02D8-   60          RTS

02D9-   A9 62       LDA   #$62         ; WRITE: das Kopieren von MAIN nach AUX
02DB-   8D ED 03    STA   $03ED          wird im MLI ($FF62) durchgeführt. Danach
02DE-   A9 FF       LDA   #$FF           Sprung zu $02D8 in AUX via XFER
02E0-   4C F6 03    JMP   $03F6        ; Sprung zu $FF62 via XFER
```

– Setzen der Pointer für den Blocktransport

```
02E3-   AD BB 03    LDA   $03BB        ; COMMAND: $01 = WRITE, $02 = READ
02E6-   4A          LSR
02E7-   B0 1D       BCS   $0306        ; Sprung für WRITE
02E9-   AD BE 03    LDA   $03BE
02EC-   85 43       STA   $43          ; READ:
02EE-   85 41       STA   $41            $003C/3D: untere Hälfte des Quell-Blocks
02F0-   AD BD 03    LDA   $03BD          $003E/3F: obere Hälfte des Quell-Blocks
02F3-   85 42       STA   $42            $0040/41: obere Speicherseite des Ziels
02F5-   85 40       STA   $40            $0042/43: untere Speicherseite des Ziels
02F7-   AD BF 03    LDA   $03BF
02FA-   85 3D       STA   $3D
02FC-   85 3F       STA   $3F
02FE-   A9 00       LDA   #$00
0300-   85 3C       STA   $3C
0302-   85 3E       STA   $3E
0304-   F0 1B       BEQ   $0321        ; „always"
0306-   AD BE 03    LDA   $03BE
0309-   85 3D       STA   $3D          ; WRITE:
030B-   85 3F       STA   $3F            $003C/3D: untere Seite des Quellbereichs
030D-   AD BD 03    LDA   $03BD          $003E/3F: obere Seite des Quellbereichs
0310-   85 3C       STA   $3C            $0040/41: obere Hälfte des Ziel-Blocks
0312-   85 3E       STA   $3E            $0042/43: obere Häflte des Ziel-Blocks
0314-   AD BF 03    LDA   $03BF
0317-   85 43       STA   $43
0319-   85 41       STA   $41
031B-   A9 00       LDA   #$00
031D-   85 42       STA   $42
031F-   85 40       STA   $40
0321-   E6 3F       INC   $3F          ; Erhöhung der Pointer für die oberen
0323-   E6 41       INC   $41            Hälften der Ziel/Quelladressen
0325-   60          RTS                ; immer mit A = 00
```

– Handler für Blocks 0/1/4/5/6 („leere Blocks")

```
0326-   20 2F 03    JSR   $032F        ; löscht $0C00/$0D00 in AUX
0329-   20 C1 02    JSR   $02C1        ; kopiert $0C00/$0D00 AUX <=> MAIN
032C-   4C DC 03    JMP   $03DC        ; Exit
```

– Löscht $0C00/$0D00 (Puffer) in AUX

```
032F-   A9 0C       LDA   #$0C         ; Zieladresse: $0C00
```

```
0331-  8D BF 03    STA   $03BF        ; Einsprung für FORMAT: A = $0E (VolDIR)
0334-  20 E3 02    JSR   $02E3        ; setzt Pointer
0337-  A8          TAY                ; = 00
0338-  91 3C       STA   ($3C),Y      ; $200 Byte des „Quellbereichs" werden
033A-  91 3E       STA   ($3E),Y        gelöscht, normalerweise $0C00..$0DFF,
033C-  88          DEY                  für FORMAT $0E00..$0FFF (VolDir-Block)
033D-  D0 F9       BNE   $0338
033F-  60          RTS
```

– Handler für Blocks $00..$5F (Blockadresse < $BFFF)

```
0340-  C9 04       CMP   #$04         ; Einsprung mit Blockno * 2: VolDIR?
0342-  D0 04       BNE   $0348        ; nein
0344-  A9 07       LDA   #$07         ; VolDIR steht auf $0E00, d.h. „Block 7"
0346-  D0 0E       BNE   $0356        ; „always"

0348-  C9 0D       CMP   #$0D         ; Blockno im Bereich $00..$06?
034A-  90 DA       BCC   $0326        ; wenn ja, Sprung

034C-  A2 00       LDX   #$00
034E-  A5 46       LDA   $46          ; Blocknummer (low)
0350-  C9 5D       CMP   #$5D         ; Blocks $5D..$5F:
0352-  90 05       BCC   $0359          Block $5D:  $1A00..$1BFF
0354-  E9 50       SBC   #$50           Block $5E:  $1C00..$1DFF
0356-  4C 83 03    JMP   $0383          Block $5F:  $1E00..$1FFF
```

– Blocknummer im Bereich $07(!), $08..$5C:

```
0359-  38          SEC                ; siehe „Blockbeschreibung"
035A-  E9 08       SBC   #$08         ; gibt Ärger mit Blockno = $07
035C-  C9 11       CMP   #$11
035E-  90 06       BCC   $0366
0360-  E9 11       SBC   #$11           Ergibt:
0362-  E8          INX
0363-  10 F7       BPL   $035C          Block $08 => $2000, Block $09 => $1000(!)
0365-  00          BRK                  Block $0A => $2200, Block $0B => $2400
0366-  A8          TAY                  Block $0C => $2600 ... Block $18 => $3E00
0367-  C0 01       CPY   #$01           Block $19 => $4000, Block $1A => $1200(!)
0369-  D0 06       BNE   $0371          Block $1B => $4200, Block $1C => $4400
036B-  8A          TXA                  Block $1D => $4600 ... Block $29 => $5E00
036C-  18          CLC                  Block $2A => $6000, Block $2B => $1400(!)
036D-  69 08       ADC   #$08           Block $2C => $6200, Block $2D => $6400
036F-  D0 12       BNE   $0383          Block $2E => $6600 ... Block $3A => $7E00
0371-  E8          INX                  Block $3B => $8000, Block $3C => $1600(!)
0372-  8A          TXA                  Block $3D => $8200, Block $3E => $400
0373-  0A          ASL                  Block $3F => $8600 ... Block $4B => 19E00
0374-  0A          ASL                  Block $4C => $A000, Block $4D => $800(!)
0375-  0A          ASL                  Block $4D => $A200, Block $4E => $400
0376-  0A          ASL                  Block $4F => $A600 ... Block $5C => 1BE00
0377-  8D BF 03    STA   $03BF          Block $5C belegt damit $BE00..$BEFF.
037A-  98          TYA
037B-  F0 02       BEQ   $037F        ; Rest = 0 => Sprung
037D-  88          DEY                ; sonst Rest - 1
037E-  98          TYA
037F-  18          CLC
0380-  6D BF 03    ADC   $03BF        ; + Divisionsergebnis * $10

0383-  0A          ASL                ; Gesamtergebnis * 2
0384-  20 BE 02    JSR   $02BE        ; setzt Pointer und kopiert MAIN <=> AUX
0387-  4C DC 03    JMP   $03DC        ; Exit => $FF44
```

– Handler für Block $03 (VBM)

```
038A-  A9 0C      LDA   #$0C          ; Zieladresse: $0C00/$0D00
038C-  8D BF 03   STA   $03BF
038F-  20 E3 02   JSR   $02E3         ; setzt Pointer und „ASL COMMAND"
0392-  B0 13      BCS   $03A7         ; Carry von „ASL", Sprung für WRITE
0394-  20 34 03   JSR   $0334         ; READ: $0C00/$0D00 wird gelöscht
0397-  A0 0F      LDY   #$0F
0399-  B9 C0 03   LDA   $03C0,Y       ; die „VBM" wird in den $0C00/$0D00-Puffer
039C-  91 3C      STA   ($3C),Y         hineinkopiert, $003C/3D von $02E3 (READ)
039E-  88         DEY                   auf $0C00/$0D00 (Quelle) gesetzt
039F-  10 F8      BPL   $0399
03A1-  20 C1 02   JSR   $02C1         ; kopiert $0C00/$0D00 nach MAIN (Zieladresse)
03A4-  4C DC 03   JMP   $03DC         ; Exit => $FF44 in MAIN

03A7-  20 C1 02   JSR   $02C1         ; WRITE: Kopie MAIN nach $0C00/$0D00 (AUX)
03AA-  20 E3 02   JSR   $02E3         ; setzt die Pointer noch einmal
03AD-  A0 0F      LDY   #$0F          ; die vom MLI „zurückgeschriebene" VBM
03AF-  B1 42      LDA   ($42),Y         wird aus dem $0C00/$0D00-Puffer kopiert,
03B1-  99 C0 03   STA   $03C0,Y         $0042/43 von $02E3 (Zieladresse) gesetzt.
03B4-  88         DEY
03B5-  10 F8      BPL   $03AF
03B7-  4C DC 03   JMP   $03DC         ; Exit => $FF44

03BA-  00                       ; INITFLAG, $FF=„bereits Formatiert"

03BB-  00                       ; COMMAND XOR $03
03BC-  00                       ; Unitnummer
03BD-  00 00                    ; Ziel/Quelladresse
03BF-  00                       ; Blocknummer (max. $7F)
```

– „VBM" einer frisch formatierten RAMDisk, Blocks $00..$07 belegt, $08..$7F frei

```
03C0- 00 FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF

03D0- F3 52 41 4D               ; Volume Directory Header für FORMAT: $F3 und „RAM"
03D4- C3 27 0D 00  00 03 00 80  ; ACCESS, Eintragslänge, aktive Einträge (00 00)
                                  Blockno VBM 00 03, TOTAL BLOCKS ($80).
                                  Das Hibyte von TOTAL BLOCKS ist $00
                                  (VolDIR vorher gelöscht).
```

**– EXIT**

```
03DC-  AD 8B C0   LDA   $C08B         ; BANKSELECT Bank 1 der LC für den Fall,
03DF-  AD 8B C0   LDA   $C08B           daß AUX Bank 2 selektiert wurde
03E2-  68         PLA                 ; Status von 80STORE
03E3-  10 05      BPL   $03EA         ; Sprung, wenn 80STORE vorher aus war
03E5-  8D 01 C0   STA   $C001         ; sonst Anschalten
03E8-  EA         NOP
03E9-  EA         NOP
03EA-  4C EF 03   JMP   $03EF
03ED-  00         BRK
03EE-  00         BRK
03EF-  A9 44      LDA   #$44          ; „Returnadresse": $FF44 in der Bank 1 der
03F1-  8D ED 03   STA   $03ED           Language Card in MAIN (MLI)
03F4-  A9 FF      LDA   #$FF

03F6-  8D EE 03   STA   $03EE         ; Einsprung für WRITE: „Return" zu $FF62
03F9-  18         CLC                 ; CLC: $0200..$BFFF Read/Write von MAIN
03FA-  B8         CLV                 ; CLV: $0000..$01FF / $D000..$FFFF von MAIN
03FB-  4C 14 C3   JMP   $C314         ; XFER: schaltet um und springt zu $FF42
03FE-  00         BRK
03FF-  00         BRK
```

# Teil VI – Updates von ProDOS

## 6.1. Die ProDOS-Version 1.0.2

Diese Version unterscheidet sich von der Version 1.0.1 nur in einigen „Kleinigkeiten".

### 6.1.1. Änderung innerhalb von REBOOT

In den Parameterblocks der von REBOOT aufgerufenen MLI-Kommandos sind die Speicherstellen geändert worden, die vom MLI sowieso überschrieben werden. Der Sinn und Zweck dieser Veränderung ist unklar.

```
12D1- 0A 80 02 00 00 FF FD FD          ; GET F'INFO, Name $0280, der Parameterblock
12E9- FD FA FD FA FA FC FC FD  FD FD      wird mit dem File Info überschrieben

12E3- 03 80 02 00 18 00                ; OPEN: Name $0280, Puffer $1800 (unverändert)
12E9- 01 00                            ; CLOSE ALL (unverändert)
12EA- 04 ** 00 20 00 00 00 00          ; READ (Start $2000, Req. Count von GET EOF)
                                         Refno von OPEN – unverändert
12F3- 02 ** FD FD FC                   ; GET EOF (Refno von OPEN), die drei letzten
                                         Bytes werden mit dem EOF überschrieben
12F8- 01 80 02                         ; GET/SET PREFIX: Name $0280 – unverändert

12FB- 00 00 00 00 00 D0                ; unbenutzt – unverändert
```

### 6.1.2. Änderung innerhalb der DISKII-RWTB

Hier wurde ein Reload der Slotnummer * 16 vor dem Abschalten des Drive-Motors eingebaut. Da für den Test auf „maximale Blocknummer überschritten" derselbe obskure Algorithmus beibehalten wurde, werden zwei Bytes Platz gebraucht. Der Programmierer hat zu diesem Zweck die Testroutine hinter PRENIBBLE verlegt und die ersten ca. $100 Bytes nach vorne verschoben. Ansonsten haben keine Änderungen stattgefunden, am logischen Ablauf der RWTB ändert sich ebenfalls nichts.
Dieser Patch hat sich ebenfalls allen Erklärungsversuchen gegenüber als außerordentlich standhaft erwiesen, d.h. er ist nach meinem Wissensstand unnötig.

```
F800-   D8          CLD
F801-   20 BE FE    JSR   $FEBE        ; Prüfung auf maximal zulässige Block-
F804-   B0 2E       BCS   $F834          nummer: exakt dieselbe Routine wie in
F806-   EA          NOP                  der Version 1.0.1 – die NOPs, die hier
F807-   EA          NOP                  für die Original-Routine stehen, belegen
```

```
F808-   EA          NOP                 zwei Byte weniger Speicherplatz.
F809-   EA          NOP
F80A-   EA          NOP                 Damit verschiebt sich der gesamte folgende
F80B-   EA          NOP                 Kode bis einschließlich $F8FE um zwei
F80C-   EA          NOP                 Bytes nach vorne, d.h. in Richtung
F80D-   EA          NOP                 niedrigerer Adressen.
F80E-   A0 05       LDY   #$05
F810-   0A          ASL
F811-   2E 56 FB    ROL   $FB56
F814-   88          DEY
F815-   D0 F9       BNE   $F810
        .....
        .....

F889-   20 DA FC    JSR   $FCDA         ; ist der Motor jetzt auf Touren?
F88C-   F0 5C       BEQ   $F8EA         ; nein, ERR=28: X enthält Slotno * 16
        .....
        .....
```

## – RETRY READ

```
F89F-   20 98 FB    JSR   $FB98         ; RDADDR, RTS mit Slotno * 16 im X-Reg
F8A2-   90 1A       BCC   $F8BE
F8A4-   CE 69 FB    DEC   $FB69
F8A7-   10 F4       BPL   $F89D
F8A9-   A9 27       LDA   #$27
F8AB-   CE 6A FB    DEC   $FB6A         ; bereits rekalibriert?
F8AE-   D0 3A       BNE   $F8EA         ; ja, I/O ERROR: X enthält Slotno * 16
        ......
        ......

F8DD-   A5 42       LDA   $42           ; COMMAND
F8DF-   4A          LSR
F8E0-   90 12       BCC   $F8F4
F8E2-   20 FD FB    JSR   $FBFD         ; READ DATAFIELD + DENIBBLE: RTS mit
F8E5-   B0 BD       BCS   $F8A4           Slotno * 16 im X-Reg
F8E7-   A9 00       LDA   #$00          ; keine Fehler
F8E9-   D0 38       BNE   $F923         ; $F8EA: „SEC" (maskiert)
F8EB-   8D 58 FB    STA   $FB58
-***********************-
F8EE-   A6 3E       LDX   $3E           ; „eingebaute" Instruktion – wozu ??
-***********************-               ; ab hier stimmen die Adressen beider
F8F0-   BD 88 C0    LDA   $C088,X         Versionen wieder überein
F8F3-   60          RTS

F8F4-   20 00 FD    JSR   $FD00         ; WRITE DATA FIELD: RTS mit Slotno * 16 in X
F8F7-   90 EE       BCC   $F8E7
F8F9-   A9 2B       LDA   #$2B          ; ERR=2B: WRITE PROTECTED
F8FB-   D0 ED       BNE   $F8EA         ; „always"
```

## – „$00"-COMMAND: TEST WPROT

```
F8FD-   A6 3E       LDX   $3E
F8FF-   BD 8D C0    LDA   $C08D,X       ; Setzen des WPROT-Tests
F902-   BD 8E C0    LDA   $C08E,X
F905-   2A          ROL                 ; setzt Carry, wenn WRITE PROTECTED
F906-   BD 8C C8    LDA   $C08C,X
F909-   4C F7 F8    JMP   $F8F7         ; Exit mit Slotno * 16 im X-Reg
        ......


FEB5-   20 85 FB    JSR   $FB85         ; das hier sind die letzten Bytes
FEB8-   AD 70 FB    LDA   $FB70           der Routine „IS SAME SLOT?"
```

```
FEBB-   D0 E9       BNE   $FEA6
FEBD-   60          RTS
```

**PRÜFROUTINE FÜR MAXIMALE BLOCKNO**

```
FEBE-   A5 46       LDA   $46          ; Blocknummer lo
FEC0-   A6 47       LDX   $47          ; hi
FEC2-   8E 56 FB    STX   $FB56        ; für spätere Division => Track/Sektorno
FEC5-   F0 09       BEQ   $FED0        ; Sprung, wenn Blockno < $100
FEC7-   CA          DEX
FEC8-   D0 04       BNE   $FECE        ; RTS mit SEC, wenn > $200
FECA-   C9 18       CMP   #$18
FECC-   90 02       BCC   $FED0        ; RTS mit CLC, wenn Blocknummer < $118
FECE-   38          SEC
FECF-   60          RTS
FED0-   18          CLC                ; ok.
FED1-   60          RTS

FED2..FEFF: unbenutzt (Nullen)
```

## 6.1.3. Änderungen innerhalb des MLI

a) Der IRQ-Handler ($D13A) verabschiedet sich nicht mehr mit ERR#01, sondern mit ERR#02, wenn sich keine der Service-Routinen für den IRQ zuständig fühlt (=> SYSTEM DEATH). Begründung: nicht ersichtlich, Dokumentation: nirgendwo.

b) Nach SET EOF wird bei Verkürzung des Files nicht nur ein SET MARK ausgeführt, um den Filepuffer wieder korrekt zu laden, sondern auch noch ein FLUSH. Dies ist die einzige Änderung, deren Grund ersichtlich ist:
Passiert nach einem SET EOF irgendein Fehler, der das System „aussteigen" läßt, enthält der File-Eintrag in der Directory jetzt die korrekte Blockanzahl, die VBM ist ebenfalls auf dem richtigen Stand. Bei der Version 1.0.1 kann man durch ein SET EOF und anschließendes Herausnehmen der Diskette Blocks belegt lassen, die überhaupt nicht mehr belegt sind – die Folge ist, daß diese Blocks bis zum nächsten FORMAT belegt bleiben.

Diese Änderung hat allerdings für jede Analyse katastrophale Folgen:
Im Gegensatz zur RWTB, wo die Änderung als Patch eingebaut wurde, verschieben sich im MLI **sämtliche** Adressen ab $E94B um 12 Byte nach hinten.
Noch schlimmer: da die Variablen offensichtlich auch einfach an das MLI angehängt wurden, verschieben sie sich ebenfalls um den gleichen Betrag.
Erst ab $F100 stimmen die Adressen wieder überein.

Zuerst die Änderung selber:

```
E945-   20 4E E8    JSR   $E84E      ; normaler Exit: setzt den neuen EOF im FCB
-***********************-
E948-   20 1C E7    JSR   $E71C      ; FLUSH
E94B-   90 07       BCC   $E954      ; Sprung, wenn kein Fehler
B94D-   AA          TAX
B94E-   68          PLA              ; das „Ergebnis" von REMOVE BLOCKS (Akku+P-Reg)
E94F-   28          PLP                liegt momentan auf dem Stack. ERR# und
E950-   8A          TXA                Carry-Status werden hier gegen den bei
E951-   38          SEC                FLUSH passierten Fehler ausgetauscht
```

```
E952-  08          PHP
E953-  48          PHA
-**********************-
E954-  68          PLA          ; beim normalen Exit: Akku + P-Reg durch
E955-  28          PLP            REMOVE BLOCKS bzw. SET MARK gesetzt.
E956-  60          RTS
```

Insgesamt sind hier also 12 Byte eingeflickt worden. Diese Änderungen haben ihre Folgen:

1. Der REBOOT-Handler hat sich um 12 Byte nach hinten verschoben. Der Relocator setzt die entsprechende Startadresse in der Global Page nicht mehr auf $EECF, sondern auf $EEDB (Speicherstelle im Relocator: $20A3).

2. Zusammen mit den Variablen-Adressen verschieben sich auch alle Tabellen. Die Tabelle, die die Startadressen der einzelnen MLI-Befehle enthält ($C0-$D3) ist erstens selber verändert, weil sich die Startadressen für die Commands GET EOF, NEWLINE, GET F'INFO, SET F'INFO, RENAME, DESTROY und SET/GET BUF verschieben. Zum anderen ist die Basis der Tabelle selber um 12 Bytes nach oben verschoben: im Relocator ist der zweite Parameterblock für „48k" entsprechend geändert: anstelle von „Relocate jedes zweiten Bytes in der Tabelle von $AF65 ab" findet diese Umsetzung „ab $AF71" statt (Speicherstellen $21C1 und $21C5).
Innerhalb derselben Parameterliste (MOVER-Block 2 für „48k") ändert sich auch die Endadresse für das Relocate des MLI selber, es wird nicht mehr der Bereich von $9000 bis $AECE, sondern der Bereich $9000-$AEDA relokalisiert (Speicherstelle $21B8).

Wenn man damit fertig ist, sich darüber zu ärgern, und akzeptiert, daß Apple, Inc. die Speicheradressen innerhalb von ProDOS mit Absicht nicht konstant hält, um ein Auseinanderpflücken dieses Betriebssytems (wie bei DOS 3.3 geschehen) zu vermeiden – dann wird man feststellen, daß die oben aufgeführten Änderungen die einzigen innerhalb der Version 1.0.2 sind.
Das soll heißen: alle Bugs sind unkorrigiert geblieben, anstelle des Bugs „STA $F052,Y" steht jetzt „STA $F05E,Y" im Error Exit von Set File Data, weil sich auch diese falsch adressierte Variable um 12 Byte nach hinten verschoben hat.

## 6.2. Kopfstand – die ProDOS-Version 1.1.1

Für den „Endbenutzer" ergeben sich bei der Version 1.1.1 zwei Unterschiede:

1) Die Version 1.1.1 ist von vorneherein nicht mehr für 48k ausgelegt, sie läuft nur noch unter 64k.
Auch wenn ich nicht so vermessen sein sollte, anzunehmen, daß die Entwickler von ProDOS die beiden Fehler im MLI, die bei 48k-Systemen Ärger machen (s. „Unterschiede zu 48k-Systemen") nicht gefunden haben – einen Grund, warum diese Version nicht mehr mit 48k laufen sollte, gibt es ansonsten nicht.

Apple, Inc. hat die „Parameterblocks für 48k" aus dem Relocator herausgenommen, der Test auf „64k" führt jetzt nicht mehr zu einem entsprechenden Setzen von MACHID alleine, sondern zu einem „Alles oder Nichts" – bei 48k (genauso wie bei einer APPLE III-Emulation) verabschiedet sich der Relocator jetzt mit „*** RELOCATION/CONFIGURATION ERROR ***" und der (bekannten) Endlosschleife.

2) Auf einem APPLE IIe (IIc) hat die RAMDisk nur noch $7F Blocks, also einen Block weniger als bei den Versionen 1.0.x.

Innerhalb der internen Logik ergeben sich folgende Unterschiede:

a) Die IRQ-Behandlung wurde erweitert, das Überschreiben der Command-Startadresse für die $C0..$D3 Commands passiert bei einem IRQ nicht mehr. Diesem Thema ist ein eigenes Kapitel gewidmet.

b) Eine „Mystery-Routine" ist hinzugekommen, sie findet sich ebenfalls im Kapitel „Interrupts der Version 1.1.1"

Die Unterschiede dieser Version sind gar nicht einmal so gravierend, wie man bei einem ersten (entsetzten) Blick auf die Listings meinem möchte: bis auf den Fehler innerhalb der IRQ-Behandlung haben sich alle anderen Bugs in diese Version hinübergerettet.

Für die erweiterte IRQ-Behandlung werden innerhalb des IRQ-Handlers im MLI 11 Bytes, für die „Mystery-Routine" werden rund $30 Bytes gebraucht. Anstatt diese Bytes auf dem bis jetzt unbenutzten Speicherplatz (z.B. $FEBE..$FEFF/ $FFEC..$FFF9) unterzubringen, hat man sich zu einem Rundschlag entschlossen.

## 6.2.1. Die Speicherbelegung unter ProDOS 1.1.1

| | |
|---|---|
| $03D0..$03D5 | Jumps des BASIC.SYSTEM (unverändert) |
| $03D6..$03E9 | Belegt mit einem Teil der „Mystery"-Routine (neu) |
| $03EA/EB | Startadresse der „Mystery"-Routine ($FD3B – neu) |
| $03F0..$03FF | BRK, RESET,„&", Ctrl-Y, NMI, IRQ.. (unverändert) |
| $BF00..$BFFF | System Global Page („offizielle" Speicherstellen unverändert) |

Der Rest der Speicherbelegung ist gründlich geändert worden:

| | |
|---|---|
| $D000..$D6FF | DISKII-RWTB (Version 1.0.2 – lediglich relokalisiert) |
| $D700..$D741 | Pathname |
| $D742..$D7BF | SET TIME (Routine nur relokalisiert – unverändert) |
| $D7C0..$D7FF | System PREFIX |

Die Reihenfolge VCB – FCB wurde vertauscht:

| | |
|---|---|
| $D800..$D8FF | FCB – Belegung und Funktion unverändert |
| $D900..$D9FF | VCB – Belegung und Funktion unverändert |
| $DA00..$DBFF | VBM Space – Funktion unverändert |
| $DC00..$DDFF | Directory Space – Funktion/Belegung unverändert |
| $DE00..$FD9A | MLI – insgesamt 15 Bytes verändert, davon 11 im IRQ-Handler |
| $FD9B..$FD64 | Zweiter Teil der „Mystery"-Routine (neu) |
| $FD65..$FE45 | Tabellen, Ende mit SYSTEM DEATH-Text |
| $FE46..$FEFF | Variablen, Reihenfolge/Bedeutung unverändert, die Speicherbereiche für den IRQ-Handler wurden verlegt |
| $FF00..$FF7F | Interface zur RAMDisk-RWTB (Platz unverändert, Routine leicht verändert) |
| $FF80..$FF9A | unbenutzt |
| $FF9B..$FFFF | IRQ/RESET/NMI (Platz und Routinen unverändert) |

## 6.2.2. Die Änderungen in der Reihenfolge der vorigen Kapitel:

1) Die Dateistrukturen (Directory-Einträge etc.) sind gleich geblieben.
2) Die Funktion und Belegung von VCB/FCB sind ebenfalls gleich, der VCB steht auf $D900 (Version 1.0.x: $F200), der FCB steht auf $D800 (Version 1.0.x: $F300).
3) Der logische Ablauf sämtlicher MLI-Befehle ist mit zwei Ausnahmen gleich geblieben:
a) OPEN prüft nicht mehr die „Min.Version" eines Files (!)
b) Nach SET EOF findet ein Aufruf von FLUSH statt (bereits seit Version 1.0.2).

4) Aufgrund der verschobenen Speicherbereiche und zwischendurch eingeflickter Bytes ergibt sich ein solcher Adressenverhau, daß das MLI noch einmal ganz gelistet wurde.
Die Variablen des MLI haben sich allesamt um $E40 nach hinten verschoben, sie sind noch einmal aufgelistet.

5) Der Boot/Relocator:
Der Boot selber ist unverändert, der Relocator ist noch einmal komplett gelistet.

6) REBOOT ist unverändert.

7) SET TIME:
Diese Routine befindet sich jetzt auf $D742.., Änderungen innerhalb der Routine selber gibt es keine.

8) Das IRQ-Handling ist erweitert worden und deshalb noch einmal besprochen.

9) Die RAMDisk-RWTB:
Diese Routine ist leicht verändert und noch einmal beschrieben.

10) Die System Global Page:
Die SGP ist, was Funktion und Belegung angeht, unverändert. Die Sprungadressen zu REBOOT, SET TIME, SYSTEM DEATH und die Einsprungadresse in das MLI sind entsprechend verändert, die Vektoren zu „NO DEVICE" und zur DISKII-RWTB in der DRIVER TABLE sind ebenfalls der neuen Speicherbelegung angepaßt.

11) Die DISKII-RWTB:
Dieser Teil ist mit der Version 1.0.2 identisch, allerdings sind die Routinen von $F800 auf $D000 gewandert. Da sich hier keine Änderungen in der Logik ergeben und die „Adress-Übersetzung" von $F800 nach $D000 gut im Kopf durchführbar ist, wurde auf ein erneutes Listing verzichtet.

## 6.2.3. Der Boot/Relocator der Version 1.1.1

Der Ablauf des Kaltstarts innerhalb des APPLE-ROM (II+) ist naturgemäß unverändert, der Ablauf innerhalb des IIe-ROMs kommt auf dasselbe heraus (Sprung zur Adresse $Cx00 auf der Disk-Controller Karte).
Neue Versionen des Programms auf der Controller-Karte sind ebenfalls (noch?) nicht bekannt.
Der Urlader des Files PRODOS ist ebenfalls unverändert.
Diese drei Sufen des Boot sind im Kapitel „Boot/Relocator" besprochen.

Innerhalb des Relocators selber gibt es einige Neuerungen:

– Weniger als 64k RAM sowie die Erkennung eines APPLE III führen zu „RELOCATION/CONFIGURATION ERROR", ein MOVER-Parameterblock für 48k existiert nicht mehr.
– Es existiert eine neue Version des F8-ROMs, in der die IRQ-Behandlung anders gelöst ist (IIc => Sprung nach $C803). Wird diese neue Version erkannt, folgt eine Kopie der IRQ-Startadresse vom ROM in die Language Card sowie in den AUX-RAM. – Des weiteren scheint eine neue Version der 80-Zeichen Karte für den IIe/IIc zu existieren, hierfür wurde noch einmal ein separater Test eingebaut (IIc enthält neuen $C3-ROM).
– Schlußendlich stimmt die Definition von MACHID nicht mehr mit den Angaben im „Technical Manual" überein, die Möglichkeit APPLE III entfällt und Bit 7 von MACHID wird durch einen anderen Test gesetzt.

Da sich durch diese zusätzlichen Tests und das Herausnehmen der Parameterblocks für ein Relocate auf 48k der gesamte folgende Programmcode kräftig verschiebt, ist der Relocator auf den folgenden Seiten noch einmal komplett gelistet.
Die „Blockbeschreibung des Boot/Relocators" (Kapitel 2.1) ist im großen und ganzen noch gültig, hier werden nur die ersten (stark veränderten) rund $260 Byte neu beschrieben und die (drei) Änderungen in den restlichen Routinen aufgelistet.

| | |
|---|---|
| $2000-$20CE | Hauptprogramm des Relocators |
| $2000 | Die momentane Unitnummer wird in den Parameterblock für ein späteres ONLINE gespeichert. |
| $2005 | TEXT/HOME und „PRODOS 1.1.1" etc (TITLE PAGE). |
| $2008 | Der erste Aufruf des MOVER ($26A6) kopiert folgende Blocks:<br>den Lader für das „.SYSTEM"-Programm auf $0800..$096B;<br>die BRK/RESET-Vektoren auf $03D0, innerhalb dieses Bereichs gibt es Änderungen (s. „Interrupts der Version 1.1.1");<br>die Prüfroutine für „128k" nach $0080.<br>Gleichzeitig werden noch $000A/B als Zeiger auf „APPLE II" gesetzt. |
| $2013 | Test, ob der Computer mindestens 48k RAM hat, wenn nicht, folgt RELOCATION/CONFIGURATION ERROR. |
| $202B | Aufruf von $2402 zum Setzen von MACHID. Es wird anhand der Speicherstelle $FBB3 im Monitor-ROM festgestellt, um welchen Typ (II,II+,IIe..) es sich handelt. Der Test auf „Raubkopie" ist unverändert, neu ist der „Ausgang" für Maschinen mit 48k RAM oder APPLE III: RTS mit gesetztem Carry. |
| $202E | Hat der Test auf $2402 ergeben, daß nur 48k RAM zur Verfügung stehen, folgt hier nicht ein Umsetzen der MOVER-Parameterblocks, sondern ein Sprung zu RELOCATION/CONFIGURATION ERROR, d.h. diese Version von ProDOS ist nicht mehr für 48k RAM vorgesehen! |
| $2030 | Der nochmalige Test von MACHID auf „64k" ist unnötig und wohl so halb von der vorigen Version stehengeblieben. |
| $2039 | Der zweite Aufruf des MOVER kopiert folgende Blocks:<br>die IRQ/RESET-Routinen und Vektoren auf $FF80..$FFFF in der LC;<br>die System Global Page auf $BF00..$BFFF;<br>der Bereich $D700..$DDFF wird gelöscht (FCB/VCB/VBM und Directory Space des MLI);<br>das MLI selber auf $DE00..$FEFF;<br>die DISKII-RWTB auf $D000..$D6FF.<br>Die Language Card (Bank 1) ist noch durch den Test auf „64k" angeschaltet. |
| $2044 | Rückschaltung auf ROM und Test des Monitor-Bytes $FBB3, für die Maschinen II und II+ geht es mit $206D weiter. |
| $204E | Für einen IIe oder IIc werden Bit 7, 6 und 3 in MACHID gelöscht, diese Bits werden durch Test der (ROM)-Speicherstelle $FBC0 neu gesetzt, man darf gespannt sein, welche neuen Monitor-Versionen für den IIe (oder welche neuen APPLEs) in Zukunft erscheinen werden. |
| $206D | Die Unitnummer des Drives, von dem aus PRODOS geladen wurde, wird als „zuletzt benutztes Unit" in die mittlerweile installierte System Global Page eingetragen.<br>Die Routine auf $252A sucht danach sämtliche Slots nach Uhrenkarte/80-Col Karte ab und setzt Bit 0/1 von MACHID entsprechend. Hierbei findet auch der Test von „APPLE II" statt.<br>Innerhalb der Slotsuch-Routine wird MACHID von $000C auf $BF98 in der System Global Page kopiert. |

$207F Die Bank 1 der Language Card wird wieder angeschaltet, der dritte Aufruf des MOVER kopiert die Routine für die Thunderclock auf $D742...$D7BE und modifiziert diesen Code danach mit der Slotnummer der Uhrenkarte (falls gefunden).

$2090 Nach einem erneuten Test von MACHID auf „64k", der wiederum von den Versionen 1.0.x stehengeblieben ist und hier immer positiv ausfällt, wird die Bank 2 der LC angeschaltet, der vierte Aufruf des MOVER kopiert REBOOT in die Bank 2 auf $D100..$D2FF. Die Bank 2 erhält das Byte $EE auf $D000 als BANKID, der REBOOT-Vektor in der System Global Page wird auf die Rück-Kopierroutine innerhalb des MLI gesetzt.

$20C2 Wenn „128k" festgestellt wurden (Test auf $2402, aufgerufen bei $202B), wird für alle Fälle auf internes $C3-ROM (anstelle des Karten-ROMs) geschaltet, danach folgt die Installation der RAMDisk-RWTB via $28-FF.

$20D1 Diese Routine ist die zweite Neuerung innerhalb des Relocators: wenn der IRQ-Vektor im Monitor-ROM auf eine Adresse unterhalb von $D0xx zeigt, wird dieser Vektor sowohl in die Language Card als auch in den AUX-RAM kopiert. Für die (bekannten) Maschinen II/II+ und IIe passiert hier nichts, der IRQ-Vektor des Monitors zeigt auf $FA86 (II) bzw. auf $FA40 (II+/IIe). Dagegen führt ein IRQ beim IIc nach $C803. Wenn dieser neue Monitor erkannt wurde, wird $DFD8 innerhalb des IRQ-Handlers im MLI auf $01 gesetzt („default": $00), das Zwischenspeichern des P-Reg und der IRQ-Returnadresse innerhalb dieses Handlers ist dann nicht mehr nötig.

$20FD Die dritte Neuerung:
Der interne ROM für den Bereich $C300..$C3FF wird angeschaltet und ROMBIT getestet, ob sich im Slot 3 eine Karte befindet, was auch bei einem IIe während der Slotsuch-Routine korrekt detektiert wird. Für die Maschinen II/II+ hat dieses Anschalten (SLOTC3ROM OFF) keine Wirkung. Wenn Slot 3 unbesetzt ist, ist die Installation von PRODOS beendet (Sprung zu $216D).
Sonst wird auf das ROM des Slots selber geschaltet, es folgt ein Test, ob sich eine (neue) 80-Zeichen Karte in Slot 3 (nicht im AUX-Slot!) befindet. Für einen IIe fällt dieser Test negativ aus, wenn die „originale" 80-Col Card installiert ist.
Für II/II+ ändert sich bei diesem Test (ID der Karte) nichts, hier wird derselbe Test durchgeführt wie in der Slotsuch-Routine.

$2128 Wenn das ID der Karte erkannt wurde, folgt ein Test von MACHID: wenn Bit 7/6 gesetzt sind und Bit 3 gelöscht ist, folgt „hat 80 Zeichen" (Setzen von Bit 1 in MACHID).

$2131 Wenn das Byte $FA der Karte den Wert $2C hat, folgt ebenfalls ein Sprung zu „hat 80 Zeichen". Für welche Karte ist dieser Test? (Beim IIc steht auf $C3FA ein $28)

$2138 Sprung hierher, wenn entweder das ID der Karte nicht stimmt oder beide Tests von $2131 negativ ausgefallen sind. Hier wird die Funktion der

80-Zeichen Karte geprüft (Test des AUX-BildspeicherRAMs). Da die Umschaltung zwischen MAIN und AUX in einem II/II+ keine Wirkung hat, fällt dieser Test für diese Maschinen immer positiv aus, d.h. jede Karte in Slot 3 wird letztendlich in MACHID als „hat 80 Zeichen" gesetzt.

$215E Fällt der Test bei $2138 negativ aus (für IIe ohne 80-Z), wird Bit 1 in MACHID gelöscht („keine 80 Zeichen").

$2165 In allen anderen Fällen wird Bit 1 in MACHID gesetzt!

$216D..$21C2 PRODOS INSTALLIERT
ONLINE, Laden der VolDIR und Sprung zum Lader für das „.SYSTEM"-Programm.

$21C3..$21FC ERROR EXIT
ROMON, HOME und „*** RELOCATION/CONFIGURATION ERROR ***"

$21FD..$2209 Parameterblocks für die MLI-Aufrufe

$220A..$226B Parameterblock-Starts und Parameterblocks für die MOVER-Aufrufe. Die ganze Prozedur des Ladens der Startadressen eines Parameterblocks ist unnötig geworden, weil nur noch Parameterblocks für 64k existieren. Ebenso ist der MOVER selber für den verwendeten Zweck reichlich überdimensioniert, ein Relocate findet nur noch für die Uhrenroutine statt. Hier werden ganze zwei Bytes neu gesetzt, wenn die Uhrenkarte nicht in Slot 1 ist. Die restlichen Programmteile von PRODOS werden lediglich kopiert.

Die Änderungen innerhalb des restlichen Code des Relocators:

$2402 WHAT MACHINE?
Wird bei dem Test nur 48k RAM oder APPLE III festgestellt, folgt RTS mit Carry set, was dann innerhalb des aufrufenden Programms ($202B) zu RELOCATION/CONFIGURATION ERROR führt.

$216F..$2649 Test auf extended 80-Col Card in der Slotsuch-Routine:
Das ID der Karte ist nicht mehr $88 auf dem Byte $0C ($C30C) der Karte, sondern $8x, da das untere Nibble vor dem Test ausgeblendet wird.

$266A/6B Die Startadresse der DISKII-RWTB, die innerhalb der Slotsuch-Routine „untergeschoben" wird, wenn ein DISKII-Controller erkannt wurde, ist nicht mehr $F800 ($B800 für 48k), sondern $D000 (nur noch 64k).

## 6.2.4. Kommentiertes Listing des RELOCATOR 1.1.1

MLI: $2D00 - $4DFF
Global Page 64k: $4E00 - $4EFF
Global Page 48k: entfällt
Thunderclock (SET TIME):$4F00 - $4F7F
IRQ/RESET: $4F80 - $4FFF
RWTB der DISKII: $5000 - $56FF
REBOOT: $5700 - $59FF
Interface RAMDisk-RWTB: $2C00 - $2C99
RAMDisk-RWTB: $2A00 - $2BFF

Weitere (temporär) kopierte Speicherbereiche:
Von $2451 auf $0080-$00C7: SET MACHID (Test auf „128k")
Von $23D8 auf $0800-$096C: Lader für das „.SYSTEM"-Programm

```
2000-   A5 43      LDA   $43          ; Slotno * 16, einziger übernommener Wert
2002-   8D FE 21   STA   $21FE        ; Unitnummer für erstes ONLINE
2005-   20 99 24   JSR   $2499        ; TITLE PAGE (HOME/„PRODOS 1.1.1...")
2008-   AE 0A 22   LDX   $220A        ; erster Aufruf des MOVER:
200B-   AC 0B 22   LDY   $220B          $226C=> $0800..$096C kopiert: Loader f. SYS
200E-   20 A6 26   JSR   $26A6          $23D8=> $03D6..$03FF kopiert: Page 3-Vecs
2011-   B0 23      BCS   $2036          „APPLE II"-Pointer auf $000A/B
2013-   A0 00      LDY   #$00           $2451 => $0080..$00C7: Test auf „128k"
2015-   A9 FF      LDA   #$FF
2017-   8D FF BF   STA   $BFFF
201A-   4D FF BF   EOR   $BFFF        ; enthält die Speicherstelle $BFFF RAM?
201D-   38         SEC
201E-   D0 6E      BNE   $208E
2020-   8D FF BF   STA   $BFFF
2023-   AD FF BF   LDA   $BFFF
2026-   D0 66      BNE   $208E        ; „RELOCATION/CONFIGURATION ERROR"
2028-   AD 82 C0   LDA   $C082
202B-   20 02 24   JSR   $2402        ; setzt MACHID ($ 000C) und $ 000A/B: „APPLE II"
202E-   B0 06      BCS   $2036        ; Sprung zu „RELOCATION/CONFIGURATION ERR"
2030-   A5 0C      LDA   $0C            für 48k-Maschinen und APPLE III
2032-   29 20      AND   #$20         ; nocheinmal Test von MACHID: hat der
2034-   D0 03      BNE   $2039          Computer mindestens 64k RAM? (unnötig)
2036-   4C C3 21   JMP   $21C3        ; nein, RELOCATION/CONFIGURATION ERROR
2039-   AE 0C 22   LDX   $220C        ; zweiter Aufruf des MOVER:
203C-   AC 0D 22   LDY   $220D        ; $4F80 => $FF80..$FFFF kopiert (IRQ)
203F-   20 A6 26   JSR   $26A6          $4E00 => $BF00..$BFFF kopiert (Global Page)
2042-   B0 4A      BCS   $208E          $D700..löschen, $2D00 => $DE00 kopiert (MLI)
2044-   AD 82 C0   LDA   $C082          $5000 => $D000..$D6FF kopiert (DISKII-RWTB)
2047-   AE B3 FB   LDX   $FBB3        ; II: $38 / II+: $EA / IIe: $06 / IIc: $06
204A-   E0 06      CPX   #$06
204C-   D0 1F      BNE   $206D        ; Sprung für II/II+

204E-   A9 E0      LDA   #$E0         ; Maschine ist IIe (IIc?)
2050-   2C C0 FB   BIT   $FBC0        ; IIe: $EA („alte Version?") / IIc: $00
2053-   08         PHP
2054-   A5 0C      LDA   $0C          ; Bit 7, 6 und 3 werden maskiert
2056-   29 37      AND   #$37         ; = %00110111
2058-   28         PLP
2059-   50 02      BVC   $205D        ; $FBC0 = $00 für IIc
205B-   30 0C      BMI   $2069          11xx xxxx: Bit 7 wird gesetzt
205D-   08         PHP                  10xx xxxx: Bit 6 und 3 werden gesetzt
205E-   09 08      ORA   #$08           01xx xxxx: Bit 3 wird gesetzt
```

```
2060-  28          PLP                  00xx xxxx: Bit 7 und 3 werden gesetzt
2061-  10 04       BPL   $2067
2063-  09 40       ORA   #$40           für einen IIe wird demnach Bit 7 in
2065-  10 04       BPL   $206B          MACHID gesetzt, Bit 6 und 3 sind gelöscht.
2067-  70 02       BVS   $206B          für einen IIc sind Bit 7 und 3 gesetzt
2069-  09 80       ORA   #$80           sowie Bit 6 gelöscht
206B-  85 0C       STA   $0C

206D-  AD FE 21    LDA   $21FE          ; Unitnummer vom Boot/„-PRODOS"-Command
2070-  8D 05 22    STA   $2205          ;
2073-  8D 30 BF    STA   $BF30          ; „zuletzt benutztes Unit"
2076-  20 2A 25    JSR   $252A          ; Absuchen der Slots nach „bekannten" Karten,
2079-  AD 05 22    LDA   $2205            dabei wird „APPLE II" geprüft
207C-  8D 30 BF    STA   $BF30          ; Reload der Unitnummer – unnötig
207F-  AD 8B C0    LDA   $C08B
2082-  AD 8B C0    LDA   $C08B          ; BANKSELECT Bank 1 der Language Card
2085-  AE 0E 22    LDX   $220E          ; dritter Aufruf des MOVER:
2088-  AC 0F 22    LDY   $220F            $4F00 => $D742..$D7BE (SET TIME), danach
208B-  20 A6 26    JSR   $26A6            Modifikation mit Slotnummer der Uhrenkarte
208E-  B0 2F       BCS   $20BF
2090-  AD 98 BF    LDA   $BF98          ; MACHID, von der Slotsuch-Routine kopiert
2093-  29 20       AND   #$20           ; falls der Computer inzwischen die
2095-  C9 20       CMP   #$20             Language Card abgeworfen hat => ERROR
2097-  D0 29       BNE   $20C2            (von der Version 1.0.2 stehengeblieben)
2099-  A9 E5       LDA   #$E5
209B-  8D 04 BF    STA   $BF04          ; der JMP zu REBOOT wird installiert
209E-  A9 FC       LDA   #$FC
20A0-  8D 05 BF    STA   $BF05
20A3-  AD 83 C0    LDA   $C083          ; BANKSELECT der Bank 2 der Language Card
20A6-  AD 83 C0    LDA   $C083            + WRITE ENABLE
20A9-  AE 10 22    LDX   $2210          ; vierter Aufruf des MOVER:
20AC-  AC 11 22    LDY   $2211            $5700 => $D100..$D3FF kopiert (REBOOT)
20AF-  20 A6 26    JSR   $26A6
20B2-  A9 EE       LDA   #$EE           ; $EE ist das BANKID für Bank 2
20B4-  8D 00 D0    STA   $D000
20B7-  AD 8B C0    LDA   $C08B          ; BANKSELECT der Bank 1
20BA-  AD 8B C0    LDA   $C08B          ; + WRITE ENABLE

20BD-  90 03       BCC   $20C2          ; „always" – Carry gelöscht von MOVER
20BF-  4C C3 21    JMP   $21C3
20C2-  AD 98 BF    LDA   $BF98          ; MACHID
20C5-  29 30       AND   #$30           ; wurden „128k" festgestellt?
20C7-  49 30       EOR   #$30
20C9-  D0 06       BNE   $20D1          ; Sprung, wenn nur 64k
20CB-  8D 0A C0    STA   $C00A          ; SLOTC3ROM: Umschalten auf internes ROM
20CE-  20 FF 28    JSR   $28FF          ; Installation der RAMDisk als „Slot 3, D2"
```

– Setzen des IRQ-Vektors – neuer Monitor-ROM bzw. IIc

```
20D1-  AD 81 C0    LDA   $C081          ; Umschalten auf „Read ROM, Write LC"
20D4-  AC FE FF    LDY   $FFFE
20D7-  AE FF FF    LDX   $FFFF          ; IRQ-Vektor im ROM
20DA-  E0 D0       CPX   #$D0           ; APPLE II/II+/IIe(alt?) X = $FA
20DC-  A9 00       LDA   #$00           ; APPLE IIc X = $C8
20DE-  B0 1D       BCS   $20FD          ; Sprung für APPLE II/II+/IIe

20E0-  AD 8B C0    LDA   $C08B
20E3-  AD 8B C0    LDA   $C08B          ; BANKSELECT der LC-Bank 1

20E6-  8D 09 C0    STA   $C009          ; ALTZP: Zpage/Stack und LC-Bereich von AUX
20E9-  8E FF FF    STX   $FFFF          ; Einsetzen des IRQ-Vektors in die
20EC-  8C FE FF    STY   $FFFE            RAMDisk (AUX 64k)
20EF-  8D 08 C0    STA   $C008          ; ALTZP: Zpage/Stack und LC-Bereich von MAIN
20F2-  8E FF FF    STX   $FFFF          ; Einsetzen des IRQ-Vektors in die
```

```
20F5-   8C FE FF    STY   $FFFE          Language Card (MLI)
20F8-   A9 01       LDA   #$01
20FA-   8D D8 DF    STA   $DFD8       ; s. „Interrupts der Version 1.1.1"
```

– Test auf 80-Zeichen – neue 80-Zeichen-Karte

```
20FD-   8D 0A C0    STA   $C00A       ; SLOTC3ROM – Umschaltung auf internes ROM
2100-   AD 99 BF    LDA   $BF99       ; ROMBIT: Ist Slot 3 besetzt?
2103-   29 08       AND   #$08
2105-   F0 66       BEQ   $216D       ; nein, Ende der Installation
2107-   8D 0B C0    STA   $C00B       ; SLOTC3ROM – Schaltung auf Slot-ROM
210A-   AD 05 C3    LDA   $C305         und Prüfung des ID der Karte. Bei einer
210D-   C9 38       CMP   #$38          Original-Karte im IIe fällt dieser Test
210F-   D0 27       BNE   $2138        negativ aus, d.h. Sprung zu $2138!
2111-   AD 07 C3    LDA   $C307
2114-   C9 18       CMP   #$18
2116-   D0 20       BNE   $2138       ID: $C305 = $38, $C307 = $18
2118-   AD 0B C3    LDA   $C30B           $C30B = $01, $C30C = $8x
211B-   C9 01       CMP   #$01        entspricht dem APPLE IIc-ROM
211D-   D0 19       BNE   $2138
211F-   AD 0C C3    LDA   $C30C
2122-   29 F0       AND   #$F0
2124-   C9 80       CMP   #$80
2126-   D0 10       BNE   $2138
```

– ID der Karte in Slot 3 stimmt

```
2128-   AD 98 BF    LDA   $BF98       ; ok. Extended 80-Col Card in Slot 3:
212B-   29 C8       AND   #$C8        ; Bit 7/6 gesetzt, Bit 3 gelöscht?
212D-   C9 C0       CMP   #$C0          welcher Rechner?
212F-   F0 34       BEQ   $2165       ; wenn ja, => „hat 80 Zeichen"
2131-   AD FA C3    LDA   $C3FA
2134-   C9 2C       CMP   #$2C        ; Original 80-Z: $00 / IIc: $28
2136-   F0 2D       BEQ   $2165       ; wenn $2C => „hat 80 Zeichen" (wer?)
```

– ID der Karte stimmt nicht oder $C3FA <> $2C (s. $2128)

```
2138-   8D 0A C0    STA   $C00A       ; SLOTC3ROM – Umschaltung auf internes ROM
213B-   8D 01 C0    STA   $C001       ; 80STORE an
213E-   8D 55 C0    STA   $C055       ; PAGE2 – Adressierung von AUX im
2141-   A9 EE       LDA   #$EE          Bildspeicherbereich. Hier wird jetzt
2143-   8D 00 04    STA   $0400         die 80-Zeichen Karte nochmal getestet,
2146-   0A          ASL                 bei einem APPLE II/II+ schaltet sich
2147-   0E 00 04    ASL   $0400          dabei natürlich nichts. Die Folge ist,
214A-   CD 00 04    CMP   $0400         daß JEDE Karte in Slot 3 als
214D-   D0 07       BNE   $2156        „80-Zeichen" in MACHID eingetragen wird.
214F-   4A          LSR
2150-   4E 00 04    LSR   $0400
2153-   CD 00 04    CMP   $0400
2156-   8D 54 C0    STA   $C054       ; PAGE2 aus – Adressierung von MAIN
2159-   8D 00 C0    STA   $C000       ; 80STORE aus
215C-   F0 07       BEQ   $2165       ; Sprung, wenn der „Test" funktioniert hat

215E-   AD 98 BF    LDA   $BF98
2161-   29 FD       AND   #$FD        ; sonst wird Bit 1 in MACHID ausgeblendet
2163-   D0 05       BNE   $216A       ; „always"

2165-   AD 98 BF    LDA   $BF98
2168-   09 02       ORA   #$02        ; setzt Bit 1 – „hat 80-Zeichen Karte"
216A-   8D 98 BF    STA   $BF98
```

## – PRODOS INSTALLIERT

```
216D-   20 00 BF    JSR   $BF00          ; ONLINE: holt den Namen des Units, von
2170-   C5 FD 21                           dem aus ProDOS gestartet wurde
2173-   D0 4E       BNE   $21C3
2175-   AD 81 02    LDA   $0281          ; Länge des Namens
2178-   29 0F       AND   #$0F
217A-   F0 47       BEQ   $21C3          ; Namenslänge = 0 => ERROR
217C-   18          CLC
217D-   69 01       ADC   #$01           ; Namenslänge + 1
217F-   8D 80 02    STA   $0280
2182-   A9 2F       LDA   #$2F           ; und „/" als erstes Zeichen
2184-   8D 81 02    STA   $0281
2187-   20 00 BF    JSR   $BF00          ; SET PREFIX mit dem Namen
218A-   C6 01 22
218D-   D0 34       BNE   $21C3
218F-   A2 00       LDX   #$00           ; Setzen von Block 00 02 (VolDIR Key)
2191-   86 14       STX   $14
2193-   A0 02       LDY   #$02
2195-   A9 0C       LDA   #$0C           ; und $0C00 als Pufferstart
2197-   85 15       STA   $15
2199-   8D 07 22    STA   $2207          ; und als Zieladresse für READ BLOCK (hi)
219C-   8C 08 22    STY   $2208          ; Blocknummer lo
219F-   8E 09 22    STX   $2209          ; Blocknummer hi
21A2-   20 00 BF    JSR   $BF00          ; READ BLOCK, das erste Mal auf $0C00
21A5-   80 04 22
21A8-   D0 19       BNE   $21C3
21AA-   A0 03       LDY   #$03
21AC-   B1 14       LDA   ($14),Y        ; Forward Reference im gerade geladenen
21AE-   AA          TAX                    Block (00 00 = Directory-Ende)
21AF-   88          DEY
21B0-   11 14       ORA   ($14),Y
21B2-   F0 0C       BEQ   $21C0          ; READ-Ende bei Forward Reference = 00 00
21B4-   B1 14       LDA   ($14),Y
21B6-   A8          TAY
21B7-   A5 15       LDA   $15
21B9-   18          CLC
21BA-   69 02       ADC   #$02           ; ansonsten Erhöhung der Zieladresse
21BC-   C9 14       CMP   #$14             bis maximal $13FF
21BE-   90 D7       BCC   $2197            und Loop (maximal 4 Blocks)

21C0-   4C 00 08    JMP   $0800          ; zum Loader für das SYSTEM-Programm
```

## – ERROR EXIT

```
21C3-   8D 82 C0    STA   $C082          ; ROMON
21C6-   20 58 FC    JSR   $FC58          ; HOME
21C9-   A0 25       LDY   #$25
21CB-   B9 D7 21    LDA   $21D7,Y        ; „** RELOCATION/CONFIGURATION ERROR **"
21CE-   99 A9 05    STA   $05A9,Y
21D1-   88          DEY
21D2-   10 F7       BPL   $21CB
21D4-   4C D4 21    JMP   $21D4          ; „Aufhängen" bis RESET

21D7- AA AA A0 D2 C5 CC CF C3  C1 D4 C9 CF CE A0 AF A0    ** RELOCATION /
21E7- C3 CF CE C6 C9 C7 D5 D2  C1 D4 C9 CF CE A0 C5 D2    CONFIGURATION ER
21F7- D2 CF D2 A0 AA AA                                   ROR **
```

## – Parameterblocks für MLI-Aufrufe

```
21FD- 02 ** 81 02              ; erstes ONLINE: Unitno auf „Boot"-Unit gesetzt
2201- 01 80 02                 ; SET PREFIX mit dem erhaltenen Namen, Puffer $0280
2204- 03 00 00 00 00 00        ; READ BLOCK, Unitnummer vom „Boot-Unit", Blockno
                                 und Zieladresse werden im Loop erhöht
```

**– Startadressen der MOVER-Parameterblocks für 64k-Maschinen**

```
220A- 12 22     ; 1. Aufruf: Loader f. SYSTEM-Programm, Page 3-Vektoren...
220C- 37 22     ; 2. Aufruf: IRQ, SGP, lösche $D700..$DEFF, MLI, DISKII-RWTB
220E- 59 22     ; 3. Aufruf: SET TIME
2210- 2F 22     ; 4. Aufruf: REBOOT
```

**– Parameterblock für den ersten Aufruf des MOVER**

```
2212- 01 00 08 6C 01 6C 22          ; Kopie von $226C nach $0800: $016C Bytes
2219- 01 D6 03 2A 00 D8 23          ; Kopie von $23D8 nach $03D6: $002A Bytes
2220- 01 0A 00 02 00 14 00          ; Kopie von $0014 nach $000A: $0002 Bytes
2227- 01 80 00 48 00 51 24          ; Kopie von $2451 nach $0080: $0048 Bytes
222E- FF                            ; STOP
```

**– Parameterblock für den vierten Aufruf**

```
222F- 01 00 D1 00 03 00 57          ; Kopie von $5700 nach $D100: $0300 Bytes
2236- FF                            ; STOP
```

**– Parameterblock für den zweiten Aufruf**

```
2237- 01 80 FF 80 00 80 4F          ; Kopie von $4F80 nach $FF80: $0080 Bytes
223E- 01 00 BF 00 01 00 4E          ; Kopie von $4E00 nach $BF00: $0100 Bytes
2245- 00 00 D7 00 07                ; Löschen des Bereichs $D700..$DEFF
224A- 01 00 DE 00 21 00 2D          ; Kopie von $2D00 nach $DE00: $2100 Bytes
2251- 01 00 D0 00 07 00 50          ; Kopie von $5000 nach $D000: $0700 Bytes
2258- FF                            ; STOP
```

**– Parameterblock für den dritten Aufruf**

```
2259- 01 42 D7 7D 00 00 4F          ; Kopie von $4F00 nach $D742: $007D Bytes
2260- 04 42 D7 69 00 42 D7 00 C1 C1 ** ; Relocate absoluter Referenzen im Bereich
                                      $C100..$C1FF um **
226B- FF                            ; STOP
```

– Letzte Stufe des Boot: **Lader für das „.SYSTEM"-Programm.** Auf $0800 kopiert!

```
226C-   A9 0C       LDA   #$0C      ; Sprung hierher mit geladener VolDIR
226E-   85 11       STA   $11
2270-   A9 04       LDA   #$04      ; Start mit erstem Eintrag nach $0C04
2272-   ** 2C **                    ; von $2270: BIT $10A5
2273-   A5 10       LDA   $10       ; d.h. „LDA $10" ist maskiert
2275-   18          CLC
2276-   6D 23 0C    ADC   $0C23     ; + Länge eines Eintrags => NEXT Eintrag
2279-   85 10       STA   $10
227B-   B0 12       BCS   $228F     ; $0010/11 auf nächsten Eintrag
227D-   6D 23 0C    ADC   $0C23     ; Test: überschreitet dieser nächste Eintrag
2280-   90 0F       BCC   $2291       die Directory-Länge, d.h. ist die
2282-   A5 11       LDA   $11         Volume-Directory hier zu Ende?
2284-   4A          LSR
2285-   90 0A       BCC   $2291     ; Speicherseite ist gerade: Blockanfang
2287-   C9 09       CMP   #$09      ; Speicherseite ungerade: $1400 erreicht?
2289-   F0 1E       BEQ   $22A9     ; „UNABLE TO FIND A .SYSTEM FILE"
228B-   A9 04       LDA   #$04      ; nächster Block: Start auf $xx04
228D-   85 10       STA   $10
228F-   E6 11       INC   $11
2291-   A0 10       LDY   #$10      ; Byte $10 im File-Entry: File Type
2293-   A9 FF       LDA   #$FF
2295-   51 10       EOR   ($10),Y
```

```
2297-  D0 DA      BNE   $2273        ; kein SYS file: NEXT
2299-  A8         TAY                ; = 0
229A-  B1 10      LDA   ($10),Y      ; Byte $00: Storage Type + Namenslänge
229C-  F0 D5      BEQ   $2273        ; NEXT, wenn File deleted
229E-  29 0F      AND   #$0F
22A0-  8D 80 02   STA   $0280        ; Namenslänge
22A3-  C9 08      CMP   #$08         ; „.SYSTEM" = 7 Zeichen
22A5-  90 CC      BCC   $2273        ; kürzer als 8 Zeichen: NEXT
22A7-  B0 03      BCS   $22AC        ; sonst Vergleich

22A9-  F0 6E      BEQ   $2319        ; „Verlängerung" für Fehler
22AB-  00         BRK

22AC-  A8         TAY                ; Namensvergleich von hinten nach vorne:
22AD-  A2 06      LDX   #$06
22AF-  B1 10      LDA   ($10),Y
22B1-  5D 65 09   EOR   $0965,X      ; „.SYSTEM" - (hier auf $23D1,X)
22B4-  0A         ASL                ; MSB wird maskiert
22B5-  D0 BC      BNE   $2273        ; endet nicht auf „.SYSTEM": NEXT
22B7-  88         DEY
22B8-  CA         DEX
22B9-  10 F4      BPL   $22AF
22BB-  A0 00      LDY   #$00         ; ok. Name endet mit „.SYSTEM"
22BD-  C8         INY
22BE-  B1 10      LDA   ($10),Y
22C0-  99 80 02   STA   $0280,Y      ; Kopie des Namens in den Parmblock mit
22C3-  09 80      ORA   #$80         ; gesetztem MSB
22C5-  99 3B 09   STA   $093B,Y      ; (hier auf $239F,Y)
22C8-  CC 80 02   CPY   $0280
22CB-  D0 F0      BNE   $22BD        ; Loop bis Namensende
22CD-  A9 A0      LDA   #$A0
22CF-  99 3C 09   STA   $093C,Y      ; ein Space wird angehängt (hier: $23A0,Y)
22D2-  98         TYA
22D3-  69 13      ADC   #$13
22D5-  8D 4F 09   STA   $094F        ; (hier auf $23BB), für „UNABLE.."-Text
22D8-  20 00 BF   JSR   $BF00
22DB-  C8 50 09                      ; OPEN: (Parmblock hier auf $23BC)
22DE-  D0 46      BNE   $2326        ; Name: $0280 / Puffer: $1400
22E0-  20 00 BF   JSR   $BF00
22E3-  D1 56 09                      ; GET EOF: (Parmblock hier auf $23C2)
22E6-  D0 3E      BNE   $2326        ; „UNABLE TO LOAD ... SYSTEM"
22E8-  AD 5A 09   LDA   $095A        ; MSByte der File-Länge (hier auf $23C6)
22EB-  D0 53      BNE   $2340        ; „SYSTEM PROGRAM TOO LARGE"
22ED-  AD 59 09   LDA   $0959        ; mittleres Byte (hier auf $23C5)
22F0-  C9 98      CMP   #$98
22F2-  B0 4C      BCS   $2340        ; „SYSTEM PROGRAM TOO LARGE"
22F4-  8D 60 09   STA   $0960        ; der EOF wird als Request Count für READ
22F7-  AD 58 09   LDA   $0958          gesetzt (2 Bytes)
22FA-  8D 5F 09   STA   $095F
22FD-  20 00 BF   JSR   $BF00
2300-  CA 5B 09                      ; READ (Parmblock hier auf $23C7)
2303-  F0 06      BEQ   $230B          Startadresse: $2000
2305-  C9 56      CMP   #$56         ; BAD BUFFER ADDRESS?
2307-  F0 37      BEQ   $2340        ; wenn ja, „SYSTEM PROGRAM TOO LARGE"
2309-  D0 1B      BNE   $2326        ; sonst „UNABLE TO LOAD ....SYSTEM"
230B-  20 00 BF   JSR   $BF00
230E-  CC 63 09                      ; keine Fehler: CLOSE (Parmblock hier $23CF)
2311-  D0 13      BNE   $2326
2313-  AD 82 C0   LDA   $C082        ; ROMON
2316-  4C 00 20   JMP   $2000        ; Sprung zum „SYSTEM"-Programm


2319-  A2 27      LDX   #$27
231B-  BD E2 08   LDA   $08E2,X      ; hier: $234E,X
```

```
231E-   9D A8 07    STA   $07A8,X        ; ** UNABLE TO FIND A „.SYSTEM" FILE **
2321-   CA          DEX
2322-   10 F7       BPL   $231B
2324-   30 25       BMI   $234B          ; „Aufhängen" bis RESET

2326-   AC 4F 09    LDY   $094F          ; hier: $23B8
2329-   A9 27       LDA   #$27
232B-   38          SEC
232C-   ED 4F 09    SBC   $094F          ; Holen des Filenamens und Zentrierung
232F-   4A          LSR                    auf dem Bildschirm (nett...)
2330-   6D 4F 09    ADC   $094F
2333-   AA          TAX
2334-   B9 2A 09    LDA   $092A,Y        ; hier: $2396,Y
2337-   9D A8 07    STA   $07A8,X        ; ** UNABLE TO LOAD xxxx.SYSTEM **
233A-   CA          DEX
233B-   88          DEY
233C-   10 F6       BPL   $2334
233E-   30 0B       BMI   $234B          ; „Aufhängen" bis RESET

2340-   A0 1E       LDY   #$1E
2342-   B9 0A 09    LDA   $090A,Y        ; hier: $2375,Y
2345-   99 AD 07    STA   $07AD,Y        ;  ** SYSTEM PROGRAM TOO LARGE **
2348-   CA          DEX
2349-   10 F7       BPL   $2342
234B-   30 FE       BMI   $234B          ; endgültiger „Aufhänger"

234D- 00

234E- AA AA A0 A0 D5 CE C1 C2  CC C5 A0 D4 CF A0 C6 C9    **  UNABLE TO FI
235E- CE C4 A0 C1 A0 A2 AE D3  D9 D3 D4 C5 CD A2 A0 C6    ND A „.SYSTEM" F
236E- C9 CC C5 A0 A0 AA AA A0                            ILE  **
2376- AA AA A0 A0 D3 D9 D3 D4  C5 CD A0 D0 D2 CF C7 D2    **  SYSTEM PROGR
2386- C1 CD A0 D4 CF CF A0 CC  C1 D2 C7 C5 A0 A0 AA AA    AM TOO LARGE  **
2396- AA AA A0 D5 CE C1 C2 CC  C5 A0 D4 CF A0 CC CF C1    ** UNABLE TO LOA
23A6- C4 A0 D8 AE D3 D9 D3 D4  C5 CD A0 AA AA AA AA AA    D X.SYSTEM *****
23B6- AA AA AA AA AA                                     *****

23BB- 00                        ; Scratch zur Mittenzentrierung des Filenamens
23BC- 03 80 02 00 14 01         ; OPEN: Pathname $0280, Puffer $1400
23C2- 02 01 00 00 00            ; GET EOF: Refno immer 1, 3 Byte Return
23C7- 04 01 00 20 00 00 00 00   ; READ BLOCK: Refno 1, Req. Count von EOF kopiert
23CF- 01 00                     ; CLOSE („ALL")

23D1- AE D3 D9 D3 D4 C5 CD                                .SYSTEM
************************** ENDE DES KOPIERTEN BEREICHS ********************
```

– diese Routine wird beim ersten Aufruf des MOVER auf $03D6 kopiert und ist in „Interrupts der Version 1.1.1" besprochen

```
23D8-   8D 08 C0    STA   $C008
23DB-   B5 42       LDA   $42,X
23DD-   8D 09 C0    STA   $C009
23E0-   95 42       STA   $42,X
23E2-   CA          DEX
23E3-   10 F3       BNE   $23D8
23E5-   A9 28       LDA   #$28           ; ERR#28: NO DEVICE CONNECTED
23E7-   38          SEC
23E8-   8D 08 C0    STA   $C008
23EB-   60          RTS

23EC- 3B FD 00 00 00 00 59 FA  59 FF 5A 4C 59 FF 4C 59
23FC- FF 4C 59 FF EB BF                  ; Page 3 – Vektoren, s. „Interrupts 1.1.1"
```

**– WHAT MACHINE?**

```
2402-  A9 00       LDA   #$00
2404-  85 0C       STA   $0C          ; MACHID auf $00
2406-  AE B3 FB    LDX   $FBB3
2409-  E0 38       CPX   #$38         ; II: MACHID $00
240B-  F0 21       BEQ   $242E
240D-  A9 80       LDA   #$80
240F-  E0 06       CPX   #$06         ; IIe + IIc: MACHID $80 (vorläufig)
2411-  F0 1B       BEQ   $242E
2413-  A9 40       LDA   #$40
2415-  E0 EA       CPX   #$EA         ; II+: MACHID $40
2417-  D0 0F       BNE   $2428        ; ERROR
2419-  AE 1E FB    LDX   $FB1E        ; Unterscheidung zum APPLE III in
241C-  E0 AD       CPX   #$AD           Emulation Mode durch PREAD (PADDLE-READ)
241E-  F0 0E       BEQ   $242E        ; ok. Ist APPLE II+
2420-  A9 D0       LDA   #$D0
2422-  E0 8A       CPX   #$8A         ; APPLE III: MACHID $D0, danach ERROR!
2424-  D0 02       BNE   $2428

2426-  38         SEC                ; => RELOCATION/CONFIGURATION ERROR!
2427-  60         RTS

2428-  A9 02       LDA   #$02         ; demoliert die erste Instr.: ($14)= $0080
242A-  91 14       STA   ($14),Y      ; Y=0
242C-  D0 20       BNE   $244E        ; „always" – Sprung in die demolierte Instr.

242E-  85 0C       STA   $0C          ; MACHID momentan
2430-  AE 8B C0    LDX   $C08B
2433-  AE 8B C0    LDX   $C08B        ; Test auf „64k": BANKSELECT Bank 1 der
2436-  A9 AA       LDA   #$AA           Language Card
2438-  8D 00 D0    STA   $D000
243B-  4D 00 D0    EOR   $D000
243E-  D0 E6       BNE   $2426        ; nur 48k: RELOCATION/CONFIGURATION ERROR!
2440-  4E 00 D0    LSR   $D000
2443-  A9 55       LDA   #$55
2445-  4D 00 D0    EOR   $D000
2448-  D0 DC       BNE   $2426        ; RELOCATION/CONFIGURATION ERROR!
244A-  A9 20       LDA   #$20
244C-  05 0C       ORA   $0C          ; ok. Bit 5 in MACHID wird gesetzt

244E-  4C 80 00    JMP   $0080        ; testet auf „128k" und setzt MACHID
```

– diese Routine wird auf $0080 kopiert

```
2451-  85 0C       STA   $0C          ; MACHID
2453-  10 35       BPL   $248A        ; für II/II+ wird der Test übersprungen
2455-  A9 EE       LDA   #$EE
2457-  8D 05 C0    STA   $C005        ; für IIe/IIc: Test auf Extended 80-Col Card
245A-  8D 03 C0    STA   $C003
245D-  8D 00 0C    STA   $0C00
2460-  8D 00 08    STA   $0800
2463-  AD 00 0C    LDA   $0C00
2466-  C9 EE       CMP   #$EE
2468-  D0 0E       BNE   $2478
246A-  0E 00 0C    ASL   $0C00
246D-  0A          ASL
246E-  CD 00 0C    CMP   $0C00
2471-  D0 05       BNE   $2478
2473-  CD 00 08    CMP   $0800
2476-  D0 03       BNE   $247B        ; ok. hat „128k"
2478-  38          SEC
2479-  B0 01       BCS   $247C        ; C gesetzt, wenn nur 64k (keine RAMDisk)
```

```
247B-   18          CLC
247C-   8D 04 C0    STA   $C004
247F-   8D 02 C0    STA   $C002
2482-   B0 06       BCS   $248A
2484-   A5 0C       LDA   $0C
2486-   09 30       ORA   #$30          ; „128k" – Bit 4/5 in MACHID werden gesetzt
2488-   85 0C       STA   $0C

248A-   A5 0B       LDA   $0B           ; Setzen von $000A/B auf $FB09
248C-   38          SEC
248D-   E9 05       SBC   #$05
248F-   85 0B       STA   $0B           ; $FB09: „APPLE II"
2491-   A5 0A       LDA   $0A
2493-   E9 00       SBC   #$00
2495-   85 0A       STA   $0A
2497-   18          CLC
2498-   60          RTS                 ; Ende des nach $0080 kopierten Bereichs
```

## – TITLE PAGE

```
2499-   AD 30 C0    LDA   $C030
249C-   8D 0C C0    STA   $C00C         ; Abschalten der extended 80-Col Card
249F-   8D 00 C0    STA   $C000
24A2-   20 84 FE    JSR   $FE84         ; SETNORM (nicht INVERSE)
24A5-   20 2F FB    JSR   $FB2F         ; INIT (Textfenster auf 40 * 24)
24A8-   20 93 FE    JSR   $FE93         ; SETVID (PR# 0)
24AB-   20 89 FE    JSR   $FE89         ; SETKBD (IN# 0)
24AE-   D8          CLD
24AF-   78          SEI
24B0-   20 58 FC    JSR   $FC58         ; HOME
24B3-   A2 07       LDX   #$07
24B5-   BD E3 24    LDA   $24E3,X       ; „APPLE //"
24B8-   9D B8 04    STA   $04B8,X
24BB-   CA          DEX
24BC-   10 F7       BPL   $24B5
24BE-   A2 16       LDX   #$16
24C0-   BD EB 24    LDA   $24EB,X       ; „PRODOS 1.1.1  18-SEP-84"
24C3-   9D B1 05    STA   $05B1,X
24C6-   CA          DEX
24C7-   10 F7       BPL   $24C0
24C9-   A2 00       LDX   #$00
24CB-   BD 02 25    LDA   $2502,X       ; „ " – 1 Leerz., dieser Loop ist für
24CE-   9D A8 06    STA   $06A8,X         zukünftige Erweiterungen reserviert
24D1-   CA          DEX
24D2-   10 F7       BPL   $24CB
24D4-   A2 26       LDX   #$26
24D6-   BD 03 25    LDA   $2503,X
24D9-   9D D0 07    STA   $07D0,X       ; „COPYRIGHT APPLE COMPUTER, INC., 1983-
84"
24DC-   CA          DEX
24DD-   10 F7       BPL   $24D6
24DF-   AD 30 C0    LDA   $C030
24E2-   60          RTS

24E3- C1 D0 D0 CC C5 A0 AF AF  D0 D2 CF C4 CF D3 A0 B1    APPLE //PRODOS 1
24F3- AE B1 AE B1 A0 A0 B1 B8  AD D3 C5 D0 AD B8 B4 A0    .1.1  18-SEP-84
2503- C3 CF D0 D9 D2 C9 C7 C8  D4 A0 C1 D0 D0 CC C5 A0    COPYRIGHT APPLE
2513- C3 CF CD D0 D5 D4 C5 D2  AC A0 C9 CE C3 AE AC A0    COMPUTER, INC.,
2523- B1 B9 B8 B3 AD B8 B4                                1983-84
```

## – Absuchen der Slots nach „bekannten" Karten (80-Z/CLOCK/CONTROLLER/ROMS)

```
252A-   A2 00       LDX   #$00
```

```
252C-  86 14     STX   $14
252E-  86 15     STX   $15
2530-  86 10     STX   $10
2532-  CA        DEX
2533-  8E 31 BF  STX   $BF31        ; DEVNUM auf $FF (keine DEVICES)
2536-  A9 C7     LDA   #$C7
2538-  85 11     STA   $11          ; $0010/11: Slotadrese, Start mit $C700
```

– NEXT SLOT

```
253A-  AD FF CF  LDA   $CFFF        ; CLRROM – Abschalten aller Zusatz-ROMs
253D-  A0 05     LDY   #$05
253F-  B1 10     LDA   ($10),Y
2541-  D9 6C 26  CMP   $266C,Y      ; DISKID: $20 xx $00 xx $03
2544-  38        SEC
2545-  D0 66     BNE   $25AD        ; Carry gesetzt: kein Disk-Controller hier
2547-  88        DEY
2548-  88        DEY
2549-  10 F4     BPL   $253F        ; ok. Disk-Controller gefunden
254B-  B1 10     LDA   ($10),Y      ; Y = $FF => $CxFF
254D-  F0 20     BEQ   $256F        ; Sprung für DISKII
254F-  C9 FF     CMP   #$FF
2551-  B0 5A     BCS   $25AD        ; Start der On-Card RWTB (lo) > $CxFE?
2553-  48        PHA                ; RWTB START – Lobyte
2554-  88        DEY
2555-  B1 10     LDA   ($10),Y      ; DEVICE CHARACTERISTICS
2557-  48        PHA
2558-  29 03     AND   #$03         ; muß mindestens „Status"
255A-  C9 03     CMP   #$03           und „Readable" sein (Bit 1/0 gesetzt)
255C-  F0 05     BEQ   $2563
255E-  68        PLA
255F-  68        PLA
2560-  38        SEC                ; irgendetwas anderes: SEC => keine Disk
2561-  D0 4A     BNE   $25AD        ; „always"
2563-  68        PLA                ; Characteristics
2564-  4A        LSR
2565-  4A        LSR
2566-  4A        LSR                ; ins untere Nibble
2567-  4A        LSR
2568-  85 12     STA   $12          ; und Speichern
256A-  4A        LSR                ; setzt Carry, wenn 2 Drives möglich
256B-  A5 11     LDA   $11
256D-  D0 0A     BNE   $2579        ; „always"

256F-  85 12     STA   $12          ; „Special" für DISKII: Characteristics = 0
2571-  AD 6A 26  LDA   $266A        ; Start der RWTB (lo) = $00
2574-  48        PHA
2575-  AD 6B 26  LDA   $266B        ; Start der RWTB (hi) $D0
2578-  38        SEC                ; Flag: 2 Drives per Controller möglich

2579-  48        PHA                ; RWTB-Startadresse hi ($Cx oder $D0)
257A-  08        PHP                ; „Push Carry" – 1/2 Drives möglich
257B-  A5 11     LDA   $11          ; Slotadresse hi ($Cx)
257D-  29 07     AND   #$07
257F-  0A        ASL
2580-  A8        TAY                ;  => Index in DEVTBL
2581-  0A        ASL
2582-  0A        ASL
2583-  0A        ASL                ; Shift ins höhere Nibble
2584-  05 12     ORA   $12          ; + Characteristics
2586-  AE 31 BF  LDX   $BF31        ; DEVNUM – Anzahl DEVICEs – 1 (Start $FF)
2589-  E8        INX
258A-  9D 32 BF  STA   $BF32,X      ; Eintrag in der DEVICE TABLE
```

```
258D-   0A          ASL                 ; 0sss 0000 => sss 0000 0
258E-   28          PLP
258F-   6A          ROR                 ; Carry von „1/2 Drives" => Dsss 0000
2590-   10 04       BPL   $2596         ; wenn gesetzt: 2 Drives möglich
2592-   E8          INX
2593-   9D 32 BF    STA   $BF32,X       ; Eintrag für Drive 2 DEVICE TABLE
2596-   8E 31 BF    STX   $BF31         ; und Setzen der neuen DEVNUM
2599-   0A          ASL                 ; bei 2 Drives ist das Carry wieder gesetzt
259A-   68          PLA
259B-   99 11 BF    STA   $BF11,Y       ; DRIVER START HIBYTE DRIVE 1
259E-   90 08       BCC   $25A8         ; Sprung, wenn nur 1 Drive möglich
25A0-   99 21 BF    STA   $BF21,Y       ; DRIVER START HIBYTE DRIVE 2
25A3-   68          PLA
25A4-   48          PHA
25A5-   99 20 BF    STA   $BF20,Y       ; LOBYTE DRIVER START DRIVE 2
25A8-   68          PLA
25A9-   99 10 BF    STA   $BF10,Y       ; LOBYTE DRIVER START DRIVE 1
25AC-   18          CLC                 ; Flag: Disk gefunden, keine weiteren Tests
```

– Sprung hierher mit SEC, wenn kein Controller gefunden

```
25AD-   20 FA 25    JSR   $25FA         ; wenn keine Disk: nur Eintrag in ROMBIT,
                                          sonst Test auf andere Karten
25B0-   C6 11       DEC   $11           ; $Cx
25B2-   A5 11       LDA   $11
25B4-   29 07       AND   #$07
25B6-   D0 82       BNE   $253A         ; NEXT Slot, bis (inklusive) Slot 1
```

– Alle Slots abgesucht

Die CHARACTERISTICS TABLE wird so sortiert, daß der Drive, von dem ProDOS geladen wurde, als höchstes DEVICE in der Table steht. Bei einem ONLINE (intern oder explizit) wird dieser Drive als erster getestet (Ausnahme: RAMDisk, sie steht immer als oberstes, d.h. erstes DEVICE in der Tabelle, weil hier die Zugriffszeit am kürzesten ist).

```
25B8-   A8          TAY                 ; = 0
25B9-   AE 31 BF    LDX   $BF31         ; DEVNUM
25BC-   BD 32 BF    LDA   $BF32,X       ; Loop abwärts durch alle DEVICEs und „Push"
25BF-   48          PHA                   aller DEVICES, die nicht im „Boot"-Slot
25C0-   29 7F       AND   #$7F            sind.
25C2-   4D 30 BF    EOR   $BF30
25C5-   0A          ASL                 ; die Drive-Nummer wird nicht berücksichtigt
25C6-   D0 02       BNE   $25CA
25C8-   68          PLA                 ; „Boot"-slot, wieder herunter vom Stack
25C9-   C8          INY                 ; Zähler für „Boot"-Slot DEVICES
25CA-   CA          DEX
25CB-   10 EF       BPL   $25BC         ; Loop
25CD-   AE 31 BF    LDX   $BF31         ; DEVNUM
25D0-   98          TYA                 ; Zähler für wieder „heruntergeworfene" D.
25D1-   F0 14       BEQ   $25E7         ; Beim Boot von nirgendwo => Restore (aua!)
25D3-   AD 30 BF    LDA   $BF30         ; = Boot slot
25D6-   9D 32 BF    STA   $BF32,X       ; Eintrag auf der obersten Position
25D9-   CA          DEX
25DA-   30 12       BMI   $25EE         ; Sprung, wenn dies das einzige DEVICE war
25DC-   88          DEY
25DD-   30 08       BMI   $25E7         ; Sprung, wenn nur ein DEVICE im „Boot"-Slot
25DF-   49 80       EOR   #$80          ; sonst wird hier fröhlich ein zweiter
25E1-   9D 32 BF    STA   $BF32,X         Drive eingetragen! (Nur „sauber" für Boot
25E4-   CA          DEX                   von einem „zwei-Drive-Controller")
25E5-   30 07       BMI   $25EE
```

```
25E7-   68          PLA                 ; Restore: wenn andere DEVICES gespeichert
25E8-   9D 32 BF    STA   $BF32,X       ; wurden, werden sie wieder in absteigender
25EB-   CA          DEX                 ; Reihenfolge in die DEVTBL eingetragen
25EC-   10 F9       BPL   $25E7

25EE-   20 7C 26    JSR   $267C         ; Prüft „APPLE II"
25F1-   F0 04       BEQ   $25F7         ; Arrgh!
25F3-   8D 98 BF    STA   $BF98         ; RTS mit MACHID im Akku => MACHID auf
25F6-   60          RTS                   der System Global Page

25F7-   4C 28 24    JMP   $2428         ; demoliert $0080 und springt dann dahin
```

– Absuchen der Slots nach anderen bekannten Karten

```
25FA-   90 5F       BCC   $265B         ; Disk gefunden: nur Eintrag in  ROMBIT
```

– Test auf Thunderclock

```
25FC-   A0 06       LDY   #$06
25FE-   B1 10       LDA   ($10),Y       ; $10/$11: $Cx00
2600-   D9 6C 26    CMP   $266C,Y       ; $08 $28 $58 $70 (PHP/PLP/CLI/BVS...)
2603-   D0 1A       BNE   $261F         ; keine Thunderclock bzw. Kompatible
2605-   88          DEY
2606-   88          DEY
2607-   10 F5       BPL   $25FE         ; ok. Thunderclock gefunden
2609-   A5 11       LDA   $11
260B-   E9 C1       SBC   #$C1
260D-   8D 6A 22    STA   $226A         ; RELOCATION FACTOR FÜR CLOCK ROUTINE
2610-   A9 4C       LDA   #$4C          ; $226A: 64k – Liste für MOVER
2612-   8D 06 BF    STA   $BF06         ; Installation des JMP in der SGP für
2615-   A5 0C       LDA   $0C             $82 – SET TIME (default: RTS)
2617-   F0 D5       BEQ   $25EE         ; MACHID jetzt – MUSS inzwischen <> 0 sein!
2619-   09 01       ORA   #$01          ; s. „Spezielles"
261B-   85 0C       STA   $0C           ; setzt: HAS CLOCK
261D-   D0 3C       BNE   $265B         ; Eintrag dieses Slots in ROMBIT
```

– Test auf Extended 80-Col Card – nicht VIDEX o.ä.!

```
261F-   A0 05       LDY   #$05
2621-   B1 10       LDA   ($10),Y
2623-   C9 38       CMP   #$38          ; Cx05: $38
2625-   D0 23       BNE   $264A         ; nein
2627-   A0 07       LDY   #$07
2629-   B1 10       LDA   ($10),Y
262B-   C9 18       CMP   #$18          ; Cx07: $18
262D-   D0 1B       BNE   $264A         ; nc
262F-   A0 0B       LDY   #$0B
2631-   B1 10       LDA   ($10),Y
2633-   C9 01       CMP   #$01          ; Cx0B: $01
2635-   D0 13       BNE   $264A         ; nor
2637-   C8          INY
2638-   B1 10       LDA   ($10),Y
263A-   29 F0       AND   #$F0
263C-   C9 80       CMP   #$80          ; Cx0C: $8x – Version 1.0.x: $88!
263E-   D0 0A       BNE   $264A         ; njet
2640-   A5 0C       LDA   $0C           ; MACHID jetzt, s. $25D4
2642-   F0 AA       BEQ   $25EE
2644-   09 02       ORA   #$02
2646-   85 0C       STA   $0C           ; setzt: HAS 80 COL CARD
2648-   D0 11       BNE   $265B         ; Eintrag dieses Slots in ROMBIT
```

– kein „bekanntes" ROM (d.h. Disk/Clock/80 Col) gefunden

```
264A-   A0 00       LDY   #$00
264C-   A2 00       LDX   #$00          ; Test dieses Slots auf konstanten READ
264E-   B1 10       LDA   ($10),Y         von $Cx00
2650-   C9 FF       CMP   #$FF          ; $FF auf Byte $00 => leerer Slot
2652-   F0 15       BEQ   $2669
2654-   D1 10       CMP   ($10),Y       ; 256 Reads von $Cx00
2656-   D0 11       BNE   $2669         ; ändert sich der Wert => leer
2658-   E8          INX                   Mit diesem Test werden auch RAM-Karten
2659-   D0 F9       BNE   $2654           in ROMBIT eingetragen
```

– Eintrag der Slotno in ROMBIT

```
265B-   A5 11       LDA   $11           ; $Cx
265D-   29 07       AND   #$07
265F-   AA          TAX
2660-   BD 74 26    LDA   $2674,X       ; $02/$04/$08/$10/$20/$40/$80 Bit Mask
2663-   0D 99 BF    ORA   $BF99
2666-   8D 99 BF    STA   $BF99         ; Bit gesetzt: ROM/RAM im Slot
2669-   60          RTS

266A- 00 D0                     ; RWTB-Startadresse für die DISKII (nur noch 64k)
266C- ** 20 ** 00 ** 03 **      ; DISKID
      08 ** 28 ** 58 ** 70      ; ID für die Thunderclock-Karte
2673- 3C                        ; viertes Byte des DISKID, nicht benutzt/geprüft
2674- 00                        ; Scratch für APPLEII-Test
2675- 02 04 08 10 20 40 80      ; Bit Mask für ROMBIT
```

**– VERIFY „APPLEII" auf $FB09...**

```
267C-   18          CLC
267D-   AC 74 26    LDY   $2674         ; = 0
2680-   B1 0A       LDA   ($0A),Y       ; $000A/0B sind auf $FB09 gesetzt
2682-   29 DF       AND   #$DF          ; Übersetzung auf Großschreibung
2684-   6D 74 26    ADC   $2674
2687-   8D 74 26    STA   $2674
268A-   2E 74 26    ROL   $2674
268D-   C8          INY
268E-   CC 77 26    CPY   $2677         ; = $08
2691-   D0 ED       BNE   $2680
2693-   98          TYA                 ; = $08
2694-   0A          ASL
2695-   0A          ASL
2696-   0A          ASL
2697-   0A          ASL
2698-   A8          TAY                 ; = $80
2699-   4D 74 26    EOR   $2674         ; muß $F5 ergeben
269C-   69 0B       ADC   #$0B          ; + $0B = 0, Carry gesetzt
269E-   D0 03       BNE   $26A3
26A0-   A5 0C       LDA   $0C           ; nach RTS: STA MACHID
26A2-   60          RTS
26A3-   A9 00       LDA   #$00          ; „Error" Exit: Setzt MACHID = 0
26A5-   60          RTS
```

**– MOVER** – kopiert und relokalisiert die einzelnen Programmteile von ProDOS

```
26A6-   86 10       STX   $10           ; Aufruf mit Startadresse der Parameterliste
26A8-   84 11       STY   $11             in X(lo) und Y(hi)
26AA-   A0 00       LDY   #$00
26AC-   B1 10       LDA   ($10),Y
```

```
26AE-  C9 05     CMP   #$05
26B0-  B0 72     BCS   $2724        ; $FF: Listenende
26B2-  AA        TAX                ; „Type"
26B3-  C8        INY
26B4-  B1 10     LDA   ($10),Y
26B6-  85 14     STA   $14
26B8-  C8        INY                ; $0014/15: Zieladresse
26B9-  B1 10     LDA   ($10),Y
26BB-  85 15     STA   $15
26BD-  C8        INY
26BE-  B1 10     LDA   ($10),Y
26C0-  85 16     STA   $16
26C2-  C8        INY                ; $0016/17: Anzahl der Bytes
26C3-  B1 10     LDA   ($10),Y
26C5-  85 17     STA   $17
26C7-  30 5D     BMI   $2726
26C9-  8A        TXA
26CA-  F0 63     BEQ   $272F        ; „Type" 0: Bereich löschen
26CC-  C8        INY
26CD-  B1 10     LDA   ($10),Y      ; sonst weitere Parameter
26CF-  85 12     STA   $12
26D1-  85 18     STA   $18
26D3-  C8        INY                ; $0012/13: Quelladresse
26D4-  18        CLC
26D5-  65 16     ADC   $16
26D7-  85 1A     STA   $1A
26D9-  B1 10     LDA   ($10),Y
26DB-  85 13     STA   $13
26DD-  85 19     STA   $19          ; $0018/19: Kopie von $0012/13
26DF-  65 17     ADC   $17
26E1-  85 1B     STA   $1B          ; $001A/1B: Obergrenze
26E3-  CA        DEX
26E4-  F0 6D     BEQ   $2753        ; „Type" 1: Kopie von „Quelle" nach „Ziel"

26E6-  8E 7F 28  STX   $287F        ; „Typen" 2/3 (Re-Set jedes 1./2. Bytes)
26E9-  C8        INY                  und 4 (Disassembly und Re-Set absoluter
26EA-  B1 10     LDA   ($10),Y        Referenzen
26EC-  8D 80 28  STA   $2880        ; Anzahl(!) der Referenzbereiche
26EF-  AA        TAX
26F0-  C8        INY
26F1-  B1 10     LDA   ($10),Y
26F3-  9D 81 28  STA   $2881,X      ; Untergrenzen der Referenzbereiche
26F6-  CA        DEX
26F7-  10 F7     BPL   $26F0
26F9-  AE 80 28  LDX   $2880
26FC-  C8        INY
26FD-  B1 10     LDA   ($10),Y
26FF-  9D 89 28  STA   $2889,X      ; Obergrenzen der Referenzbereiche
2702-  CA        DEX
2703-  10 F7     BPL   $26FC
2705-  AE 80 28  LDX   $2880
2708-  C8        INY
2709-  B1 10     LDA   ($10),Y
270B-  9D 91 28  STA   $2891,X      ; Additionsfaktoren für den Fall, daß
270E-  CA        DEX                  die Referenz im Bereich liegt. Subtraktion
270F-  10 F7     BPL   $2708          erfolgt durch Overflow! (8-Bit Addition)
2711-  20 59 27  JSR   $2759        ; erhöht den Parameter-Pointer
2714-  AE 7F 28  LDX   $287F
2717-  E0 03     CPX   #$03         ; war das „Type 4"?
2719-  F0 0E     BEQ   $2729
271B-  20 BD 27  JSR   $27BD        ; ansonsten „Type 3", setzt jedes 2. Byte
271E-  20 66 27  JSR   $2766        ; danach Umkopieren – nicht benutzt
2721-  4C AA 26  JMP   $26AA        ; nächster Parametersatz in der Liste
2724-  18        CLC
```

```
2725-  60          RTS                ; Exit bei Ende der Parameterliste
2726-  4C F3 27    JMP   $27F3
2729-  20 CF 27    JSR   $27CF        ; „Type 4" Handler: Disasm und Re-Set
272C-  4C 1E 27    JMP   $271E        ; danach Umkopieren - nicht genutzt
```

– Type 0: Löschen eines Bereichs

```
272F-  20 59 27    JSR   $2759        ; setzt Pointer auf nächsten Parametersatz
2732-  A9 00       LDA   #$00
2734-  A4 17       LDY   $17          ; $0016/17: Anzahl der Bytes
2736-  F0 0C       BEQ   $2744        ; keine ganzen Seiten mehr übrig
2738-  A8          TAY
2739-  91 14       STA   ($14),Y      ; $0014/15: Zieladresse
273B-  C8          INY
273C-  D0 FB       BNE   $2739        ; Loop für $100 Bytes
273E-  E6 15       INC   $15          ; Zieladresse + $100
2740-  C6 17       DEC   $17          ; Seitenzähler minus 1
2742-  D0 F5       BNE   $2739
2744-  A4 16       LDY   $16
2746-  F0 08       BEQ   $2750        ; keine einzelnen Bytes mehr übrig
2748-  A8          TAY
2749-  91 14       STA   ($14),Y
274B-  C8          INY
274C-  C4 16       CPY   $16          ; Vergleich mit Anzahl - Lobyte
274E-  90 F9       BCC   $2749
2750-  4C AA 26    JMP   $26AA        ; nächster Parametersatz aus der Liste
```

– Type 1 „Handler"

```
2753-  20 59 27    JSR   $2759        ; Parameter-Pointer auf nächsten Satz
2756-  4C 1E 27    JMP   $271E        ; ruft dann die Kopier-Routine auf
```

– Setzt den Pointer in der Parameterliste auf den nächsten Satz

```
2759-  98          TYA                ; Zähler und momentaner Index, steht
275A-  38          SEC                  im Moment auf dem letzten bereits
275B-  65 10       ADC   $10            erfaßten Parameter, deshalb Addition
275D-  85 10       STA   $10            mit SEC (+1).
275F-  A5 11       LDA   $11          ; $0010/11: Pointer zum Parametersatz
2761-  69 00       ADC   #$00
2763-  85 11       STA   $11
2765-  60          RTS
```

– Type 1: Kopier-Routine

```
2766-  A5 13       LDA   $13          ; $0012/13: Quelle
2768-  C5 15       CMP   $15          ; $0014/15: Ziel
276A-  90 0B       BCC   $2777
276C-  D0 2C       BNE   $279A          wenn Zieladresse < Quelladresse,
276E-  A5 12       LDA   $12            wird in aufsteigender Reihenfolge kopiert
2770-  C5 14       CMP   $14
2772-  90 03       BCC   $2777
2774-  D0 24       BNE   $279A
2776-  60          RTS                ; RTS, wenn Quelle und Ziel identisch sind

2777-  A4 17       LDY   $17          ; Ziel > Quelle:
2779-  98          TYA
277A-  18          CLC
277B-  65 13       ADC   $13          ; Kopie in absteigender Reihenfolge:
277D-  85 13       STA   $13            Zuerst werden beide Pointer um die
277F-  98          TYA                  Anzahl der zu kopierenden Speicherseiten
2780-  18          CLC                  erhöht.
```

```
2781-  65 15     ADC   $15
2783-  85 15     STA   $15
2785-  A4 16     LDY   $16          ; danach Start mit einzelnen Bytes
2787-  F0 08     BEQ   $2791        ; wenn auf Seitengrenze, DEC des Zählers(hi)
2789-  88        DEY
278A-  B1 12     LDA   ($12),Y      ; Kopieren
278C-  91 14     STA   ($14),Y
278E-  98        TYA
278F-  D0 F8     BNE   $2789
2791-  C6 15     DEC   $15          ; danach werden die ganzen Seiten kopiert
2793-  C6 13     DEC   $13
2795-  C6 17     DEC   $17          ; Herunterzähler für ganze Seiten
2797-  10 F0     BPL   $2789
2799-  60        RTS

279A-  A0 00     LDY   #$00         ; Zieladresse < Quelle: aufsteigende Folge
279C-  A5 17     LDA   $17          ; Sprung, wenn keine ganzen Seiten
279E-  F0 0F     BEQ   $27AF          zu kopieren sind
27A0-  B1 12     LDA   ($12),Y
27A2-  91 14     STA   ($14),Y      ; sonst die ganzen Seiten zuerst
27A4-  C8        INY
27A5-  D0 F9     BNE   $27A0
27A7-  E6 15     INC   $15
27A9-  E6 13     INC   $13
27AB-  C6 17     DEC   $17          ; Herunterzähler für ganze Seiten
27AD-  D0 F1     BNE   $27A0
27AF-  A5 16     LDA   $16          ; einzelne Bytes übrig?
27B1-  F0 09     BEQ   $27BC
27B3-  B1 12     LDA   ($12),Y      ; Kopie einzelner Bytes
27B5-  91 14     STA   ($14),Y
27B7-  C8        INY
27B8-  C4 16     CPY   $16          ; Vergleich mit Byte-Anzahl (lo)
27BA-  D0 F7     BNE   $27B3
27BC-  60        RTS
```

– Type 2/3: Re-Set jedes Nten Bytes. N = Type - 1 (=> $287F)

```
27BD-  AC 7F 28  LDY   $287F        ; N
27C0-  88        DEY
27C1-  B1 18     LDA   ($18),Y      ; Pointer auf den Code
27C3-  20 FB 27  JSR   $27FB        ; Test auf Bereich(e) und ggf. Re-Set
27C6-  AD 7F 28  LDA   $287F        ; N
27C9-  20 17 28  JSR   $2817        ; $0018/19 plus N, Vergleich mit Obergrenze
27CC-  90 EF     BCC   $27BD        ; Loop bis zur Obergrenze
27CE-  60        RTS
```

– Type 4: Disassembly und Korrektur absoluter Referenzen

```
27CF-  A0 00     LDY   #$00
27D1-  B1 18     LDA   ($18),Y      ; Pointer auf den Code
27D3-  20 2A 28  JSR   $282A        ; berechnet die Länge einer Instruktion
27D6-  F0 11     BEQ   $27E9        ; ERROR: Ungültiger Opcode
27D8-  C9 03     CMP   #$03         ; 3-Byte-Instruktion, d.h. absolute Ref?
27DA-  D0 07     BNE   $27E3        ; nein
27DC-  A0 02     LDY   #$02         ; das 02. Byte wird geprüft
27DE-  20 FB 27  JSR   $27FB          und ggf. korrigiert
27E1-  A9 03     LDA   #$03         ; noch einmal: Länge der Instruktion
27E3-  20 17 28  JSR   $2817        ; $0018/19 + Instr.Länge, Test auf Obergrenze
27E6-  90 E7     BCC   $27CF        ; Loop, wenn Obergrenze noch nicht erreicht
27E8-  60        RTS

27E9-  68        PLA                ; ERR EXIT – einzige Fehlermöglichkeit im
27EA-  68        PLA                  MOVER – ungültiger Opcode bei Type 4
```

```
27EB-  A6 18     LDX  $18          ; diese Daten werden nicht verwertet,
27ED-  A4 19     LDY  $19
27EF-  A9 FF     LDA  #$FF         ; dieser Wert auch nicht
27F1-  38        SEC               ; RTS zur Routine, die den MOVER aufgerufen
27F2-  60        RTS                 hat => „RELOCATION/CONFIGURATION ERROR"

27F3-  A6 10     LDX  $10          ; nicht benutzt und wohl von einer
27F5-  A4 11     LDY  $11            früheren Version stehengeblieben
27F7-  A9 00     LDA  #$00           (s. „Spezielles")
27F9-  38        SEC
27FA-  60        RTS
```

– Korrektur jedes Nten Bytes, falls im Referenz-Zielbereich

```
27FB-  B1 18     LDA  ($18),Y      ; Aufruf mit n in Y-Reg
27FD-  AE 80 28  LDX  $2880        ; kann > 1 sein!
2800-  DD 81 28  CMP  $2881,X      ; Untergrenze
2803-  90 07     BCC  $280C        ; außerhalb des Referenz-Zielbereichs
2805-  DD 89 28  CMP  $2889,X      ; Obergrenze
2808-  90 06     BCC  $2810        ; ok. Innerhalb des Zielbereichs
280A-  F0 04     BEQ  $2810
280C-  CA        DEX
280D-  10 F1     BPL  $2800        ; Loop durch alle gesetzten Bereiche
280F-  60        RTS
2810-  18        CLC
2811-  7D 91 28  ADC  $2891,X      ; Addition (mit Overflow: Subtraktion) des
2814-  91 18     STA  ($18),Y        Korrekturfaktors für diesen Zielbereich
2816-  60        RTS
```

– Erhöht $0018/19 und testet gegen Obergrenze

```
2817-  18        CLC               ; Aufruf mit Instr.Länge bzw. Schritt-
2818-  65 18     ADC  $18            weite im Akku
281A-  A4 19     LDY  $19
281C-  90 01     BCC  $281F
281E-  C8        INY               ; Erhöhung des Zielpointers (hi)
281F-  C4 1B     CPY  $1B
2821-  90 02     BCC  $2825
2823-  C5 1A     CMP  $1A          ; $001A/1B: Obergrenze
2825-  85 18     STA  $18
2827-  84 19     STY  $19          ; neue Adresse wird immer gesetzt
2829-  60        RTS               ; mit SEC, wenn Obergrenze ereicht
```

– Bestimmung der Länge einer Instruktion

```
282A-  48        PHA               ; Aufruf mit dem ersten Instruktions-Byte
282B-  29 03     AND  #$03           im Akku
282D-  A8        TAY               ; Bereich: 00-03
282E-  68        PLA
282F-  4A        LSR
2830-  4A        LSR
2831-  AA        TAX
2832-  BD 3F 28  LDA  $283F,X      ; Tabellenbereich: $00-$3F
2835-  88        DEY
2836-  30 04     BMI  $283C
2838-  4A        LSR
2839-  4A        LSR
283A-  D0 F9     BNE  $2835
283C-  29 03     AND  #$03         ; 00: ungültiger Opcode, 1-3: Länge
283E-  60        RTS
```

### – Tabelle zur Bestimmung der Instruktionslängen

```
283F- 09 28 19 3C 0A 28 0D 3C  0B 2A 19 3F 0A 28 0D 3C
284F- 09 28 19 3F 0A 28 0D 3C  09 28 19 3F 0A 28 0D 3C
285F- 08 2A 11 3F 0A 2A 1D 0C  2A 2A 19 3F 0A 2A 1D 3F
286F- 0A 2A 19 3F 0A 28 0D 3C  0A 2A 19 3F 0A 28 0D 3C

287F- 00                       ; N (Schrittweite) für Typen 2/3 (Relocate)
2880- 00                       ; Anzahl der Referenze-Bereiche
2881- 00 00 00 00 00 00 00 00  ; Platz für Untergrenzen
2889- 00 00 00 00 00 00 00 00  ; Platz für Obergrenzen
2891- 00 00 00 00 00 00 00 00  ; Korrekturwerte - selber Index wie Bereichsgrenzen

2899- 00 00 00 00 00 00 00
28A0- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
28B0- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
28C0- 00 00 00 00                          00 00 00 00 00
28D0- 00 00 00 00     nicht benutzt        00 00 00 00 00
28E0- 00 00 00 00                          00 00 00 00 00
28F0- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
```

### – Installation der RAMDisk-RWTB

```
28FF-   EA          NOP
2900-   A0 99       LDY   #$99
2902-   B9 00 2C    LDA   $2C00,Y
2905-   99 00 FF    STA   $FF00,Y       ; das Interface zur RAMDisk-RWTB wird
2908-   88          DEY                   von $2C00 nach $FF00..$FF99 kopiert
2909-   C0 FF       CPY   #$FF
290B-   D0 F5       BNE   $2902

290D-   A2 00       LDX   #$00
290F-   86 3C       STX   $3C
2911-   CA          DEX
2912-   86 3E       STX   $3E
2914-   A2 2A       LDX   #$2A          ; Setup für Block-Transport:
2916-   86 3D       STX   $3D
2918-   E8          INX                   $003C/3D: $2A00 (Startadresse)
2919-   86 3F       STX   $3F
291B-   A9 00       LDA   #$00            $003E/3F: $2BFF (Endadresse)
291D-   85 42       STA   $42
291F-   A9 02       LDA   #$02            $0042/43: $0200 (Zieladresse)
2921-   85 43       STA   $43
2923-   38          SEC
2924-   20 11 C3    JSR   $C311         ; kopiert $2A00..$2BFF auf AUX $0200..$03FF

2927-   A9 00       LDA   #$00
2929-   8D 26 BF    STA   $BF26         ; RWTB-Start für RAMDisk Driver ($FF00)
292C-   A9 FF       LDA   #$FF            wird in die DRIVER TABLE eingetragen
292E-   8D 27 BF    STA   $BF27
2931-   EE 31 BF    INC   $BF31         ; DEVNUM wird um eins erhöht
2934-   AE 31 BF    LDX   $BF31
2937-   A9 BF       LDA   #$BF          ; und die RAMDisk als „Slot 3, Drive 2"
2939-   9D 32 BF    STA   $BF32,X         in die DEVICE TABLE eingetragen
293C-   60          RTS

293D..29FF      unbenutzt (Nullen)

2A00..2BFF      RAMDisk-RWTB: auf AUX $0200..$03FF kopiert

2C00..2C99      Interface zur RAMDisk-RWTB: auf $FF00 kopiert
```

```
2C9A..2CFF      unbenutzt (Nullen)

2D00..4DFF      MLI: auf $DE00.. kopiert (Bank 1)

4E00..4EFF      System Global Page, auf $BF00..$BFFF kopiert

4F00..4F7F      SET TIME: auf $D142..$D1C1 (Bank 1) kopiert, danach Relocate auf den
                Slot, in dem die Thunderclock-Karte gefunden wurde

4F80..4FFF      IRQ/RESET: auf $FF80..$FFFF kopiert

5000..56FF      RWTB für die DISKII: auf $D000..$D6FF kopiert (Bank 1)

5700..59FF      REBOOT: auf $D100..$D2FF kopiert (Bank 2)
```

Der File PRODOS der Version 1.1.1 endet mit der Adresse $59FF.

## 6.2.5. Blockbeschreibung des MLI der Version 1.1.1

Um dieses Buch nicht noch dicker zu machen, als es sowieso schon ist, werden hier nur die Unterschiede zur Blockbeschreibung der Version 1.0.1 aufgeführt. Die dort gegebene Beschreibung ist in weiten Teilen gültig, bei den angegebenen Adressen muß jeweils rund $0E00 dazuaddiert werden.

$DE00..$DF39 $ Einsprung in das MLI, Verteilung auf die verschiedenen COMMAND-Arten (REBOOT, SET TIME, $4x, $8x, $Cx..$D3), Aufruf der Command Handler, Exit, NO DEVICE, DO DISK I/O und ALLOC/DEALLOC INT.
Hier ist außer den Tabellenadressen und Subroutinen-Starts alles gleich geblieben, der „Offset" beträgt exakt $0E00.

$DF3A..$DFE4 IRQ-Handler
Hier sind insgesamt 11 Bytes eingebaut worden, die vom Relocator gesetzte Speicherstelle $DFD8 (<> 00 bei „neuem Monitor") wird getestet. Falls diese Speicherstelle gesetzt ist, findet ein Zwischenspeichern des P-Reg und der IRQ-Returnadresse sowie das Zurückladen dieser Werte nach dem IRQ-Service nicht statt.
Der Offset beträgt ab hier $0E0B.

$DFE5..$E7FF SET SYSERR, SYSTEM DEATH, $Cx..$D3 COMMAND HANDLER, Copy & Verify Pathname, SET PREFIX, GET PREFIX, Set File Data, ONLINE, CREATE, ENTRY & UPDATE, GET FILE, FIND FILE ENTRY, Search Volume.
Innerhalb dieser Routinen hat keine Änderung stattgefunden, der Offset betragt durchgehend $0E0B.

$E800..$E863 Wenn innerhalb von „Search Volume" der Volume-Name nicht im VCB gefunden wurde oder die Disk gewechselt wurde, findet ein internes ONLINE statt (unverändert).

$E830 Nur für den Fall, daß die Disk gewechselt wurde, wird zuerst auf dem „alten" Unit die VolDIR gelesen. Zu diesem Zweck existiert ein Loop innerhalb der kopierten DEVICE TABLE, der in den Versionen 1.0.x mit BPL endet.
Hier findet sich die nächste eingeflickte Instruktion: für den Fall, daß die DEVICE TABLE nur ein einziges Unit enthält, ist an das Ende des Loop ein zusätzliches „BMI" zu ERR#45: VOLUME NOT FOUND eingebaut worden.
Damit verändert sich der Offset des folgenden Codes auf $0E0D.

$E864..$EEC5 Der Rest der Suchroutinen nach einem File-Eintrag, SPACE ON VOLUME, DEALLOCATE BLOCK, ALLOCATE NEXT BLOCK, die Block-R/W Routinen innerhalb des DIR- und VBM-Bereichs, GET MARK, SET MARK und die Block-R/W Routinen innerhalb eines Filepuffers.
In diesem Bereich haben keine Änderungen stattgefunden, der Offset beträgt $0E0D.

$EEC6..$EFB2 OPEN

$EF23 Nach dem Laden eines File-Eintrags wurde hier die „Min. Version" des Files gegen die MLI-Version geprüft. Dieser Test ist entfallen, aus „CMP (MLI-Version) / LDA #00 / BCS o.k." wurde „LDA #00 / BEQ o.k."
Damit verändert sich der Offset zum folgenden Code auf $0E0A.

$EFB3..$F610 SCAN FCB, READ, WRITE, „ACTIVATE", CLOSE und FLUSH.
Keine Änderungen, der Offset beträgt $0E0A.

$F611..$F760 SET EOF

$F752 Diese Erweiterung existiert bereits in der Version 1.0.2: nach Verkürzung eines Files durch SET EOF wird ein FLUSH durchgeführt, um den File-Eintrag und die VBM des Volumes auf den neuesten Stand zu bringen. Falls innerhalb dieses FLUSH ein Fehler auftritt, wird die ERR# und das P-Register mit gesetztem Carry auf den Stack gebracht (Austausch mit einer eventuell vorher gesetzten ERR# von REMOVE BLOCKS oder SET MARK).
Diese Erweiterung benötigt 12 Bytes, der Offset beträgt damit $0E16.

$F761..$FD3A GET EOF, NEWLINE, GET FILE INFO, SET FILE INFO, RENAME, DESTROY, REMOVE BLOCKS, die Test-Routinen für die System Bit Map, DEALLOCATE FILEBUFFER, SET/GET BUFFER und die Kopier-Routine für REBOOT.

Innerhalb dieser Routinen hat keine Änderung stattgefunden, der Offset beträgt $0E16.

$FD3B..$FD64 Der erste Teil der „Mystery Routine" ist $2A Bytes lang. Die Routine selber ist in „Interrupts der Version 1.1.1" beschrieben.
Der Offset zu den folgenden Tabellen beträgt damit genau $0E40.

$FD65..$FE45 Tabellen des MLI und Text von SYSTEM DEATH, unverändert.

$FE46..FEFF Variablenbereich des MLI. Die Speicherbereiche für den IRQ-Handler sind verlegt worden. (s. „MLI-Variablen der Version 1.1.1")

$FF00..$FF7F Interface zur RAMDisk-RWTB

$FF80..$FF9B unbenutzt

$FF9B..$FFFF IRQ/RESET/NMI-Handler – unverändert (s. „Interrupts/RESET").

## 6.2.6. Das MLI der Version 1.1.1

Der Einsprung in das MLI erfolgt von der System Global Page via JMP (nicht JSR!). Alle MLI-Befehle enden via $DE78 mit MLIEXIT ($BFA0). Ausnahmen: REBOOT, der INT-Service und SYSTEM DEATH während der Befehlsausführung (oder während des INT-Service).

```
DE00-   D8          CLD             ; das BANKID der Version 1.0 steht noch hier
DE01-   8C 9F BF    STY   $BF9F     ; YSAVE
DE04-   8E 9E BF    STX   $BF9E     ; XSAVE
DE07-   68          PLA             ; Returnadresse zum „user"-Programm
DE08-   85 40       STA   $40
DE0A-   18          CLC
DE0B-   69 04       ADC   #$04      ; +4 => COMMAND und Adresse des Parmblock
DE0D-   8D 9C BF    STA   $BF9C       werden übersprungen (3 Bytes + RTI)
DE10-   68          PLA
DE11-   85 41       STA   $41       ; MLIRTS in der System Global Page
DE13-   69 00       ADC   #$00
DE15-   8D 9D BF    STA   $BF9D
DE18-   A0 00       LDY   #$00      ; SYSERR = 0. Wird bei Fehler während der
DE1A-   8C 0F BF    STY   $BF0F       Befehlsausführung gesetzt und beim Exit
DE1D-   C8          INY               geladen (0 = kein Fehler)
DE1E-   B1 40       LDA   ($40),Y   ; COMMAND
DE20-   4A          LSR             ; CMD + hinibble CMD =>
DE21-   4A          LSR             ; $40=>$44 $41=>$45 $81=>$89 $82=>$8A
DE22-   4A          LSR             ; $C0=>$CC $C1=>$CD ....
DE23-   4A          LSR
DE24-   18          CLC
DE25-   71 40       ADC   ($40),Y
DE27-   29 1F       AND   #$1F      ; Reduktion auf $00-$1F
DE29-   AA          TAX             ; Index in der MLI-eigenen COMMAND Table
DE2A-   B1 40       LDA   ($40),Y   ; COMMAND aus dem „user"-Programm
DE2C-   DD 65 FD    CMP   $FD65,X   ; Überprüfung
DE2F-   D0 76       BNE   $DEA7     ; ERR #1: BAD SYSTEM CALL NUMBER
DE31-   C8          INY
DE32-   B1 40       LDA   ($40),Y   ; Pointer zum Parmblock des „user"-Programms
DE34-   48          PHA               wird auf $0040/41 kopiert
```

```
DE35-  C8          INY
DE36-  B1 40       LDA   ($40),Y
DE38-  85 41       STA   $41
DE3A-  68          PLA
DE3B-  85 40       STA   $40
DE3D-  A0 00       LDY   #$00
DE3F-  BD 85 FD    LDA   $FD85,X      ; Tabelle mit der Anzahl der Parameter
DE42-  F0 1C       BEQ   $DE60        ; Sprung für SET TIME ($82): 0 Parameter
DE44-  D1 40       CMP   ($40),Y      ; ansonsten Vergleich
DE46-  D0 63       BNE   $DEAB        ; ERR #4: BAD SYSTEM CALL PARAMETER COUNT
DE48-  BD 65 FD    LDA   $FD65,X      ; reload CMD (diesmal aus der MLI-Table)
DE4B-  C9 65       CMP   #$65
DE4D-  F0 0E       BEQ   $DE5D        ; REBOOT
DE4F-  0A          ASL
DE50-  10 14       BPL   $DE66        ; $8x Commands (außer $82)
DE52-  B0 1D       BCS   $DE71        ; $Cx,$Dx Commands
DE54-  4A          LSR
DE55-  29 03       AND   #$03         ; $4x-Commands bleiben übrig
DE57-  20 F3 DE    JSR   $DEF3        ; $4x COMMAND HANDLER
DE5A-  4C 78 DE    JMP   $DE78        ; Exit
DE5D-  4C 03 BF    JMP   $BF03        ; REBOOT COMMAND HANDLER (kein Return)
DE60-  20 06 BF    JSR   $BF06        ; $82 - Sprung zu SET TIME oder RTS
DE63-  4C 78 DE    JMP   $DE78        ; Exit
```

### – $80/$81 Command Handler – Vorstufe

```
DE66-  4A          LSR
DE67-  69 01       ADC   #$01
DE69-  85 42       STA   $42          ; $80=>$01  $81=>$02
DE6B-  20 B2 DE    JSR   $DEB2        ; eigentlicher COMMAND HANDLER
DE6E-  4C 78 DE    JMP   $DE78        ; Exit
```

### – $Cx, $Dx Command Handler – Vorstufe

```
DE71-  4A          LSR
DE72-  29 1F       AND   #$1F         ; Bereich: $00-$13 für $C0-$D3
DE74-  AA          TAX
DE75-  20 47 E0    JSR   $E047        ; eigentlicher COMMAND HANDLER
```

### – EXIT ROUTINE

Xref: JP $DE5A / JP $DE63 / JP $DE6E – Exits der COMMAND HANDLER

```
DE78-  A9 00       LDA   #$00
DE7A-  8D 95 BF    STA   $BF95        ; löscht „Backup" flag
DE7D-  AC 0F BF    LDY   $BF0F        ; SYSERR (ERR#)
DE80-  C0 01       CPY   #$01         ; setzt Carry wenn <> 0
DE82-  98          TYA
DE83-  08          PHP
DE84-  78          SEI
DE85-  4E 9B BF    LSR   $BF9B        ; setzt MLIAKTIV zurück
DE88-  68          PLA                ; P-reg (Carry!)
DE89-  AA          TAX
DE8A-  AD 9D BF    LDA   $BF9D        ; Reload der Returnadresse
DE8D-  48          PHA                  zum „user"-Programm
DE8E-  AD 9C BF    LDA   $BF9C
DE91-  48          PHA
DE92-  8A          TXA                ; P-reg (Carry)
DE93-  48          PHA
DE94-  98          TYA
DE95-  AE 9E BF    LDX   $BF9E        ; XSAVE
DE98-  AC 9F BF    LDY   $BF9F        ; YSAVE
```

```
DE9B-   48          PHA             ; ERR# (0 = kein Fehler)
DE9C-   AD F4 BF    LDA   $BFF4     ; BANKSEL – welche Bank war vorher an?
DE9F-   4C A0 BF    JMP   $BFA0     ; MLIEXIT
```

MLIEXIT schaltet auf die Konfiguration zurück (Bank/ROM), die vor dem MLI-Aufruf aktiv war. Die ERR# wird vom Stack geholt und ein RTI zum „user"-Programm ausgeführt. Bei dem RTI wird das P-Reg ebenfalls vom Stack geholt.

**– NO DEVICE**

Alle „unbesetzten" RWTB-Startadressen in der DRIVER TABLE zeigen hierher.

```
DEA2-   A9 28       LDA   #$28      ; ERR #28: NO DEVICE CONNECTED
DEA4-   20 09 BF    JSR   $BF09     ; SET SYSERR – RTS zur Routine, die DEDA
                                      aufgerufen hat
DEA7-   A9 01       LDA   #$01
DEA9-   D0 02       BNE   $DEAD     ; ERR #01: BAD SYSTEM CALL NUMBER
DEAB-   A9 04       LDA   #$04      ; ERR #04: BAD SYSTEM CALL PARAMETER COUNT
DEAD-   20 D7 DE    JSR   $DED7     ; SET SYSERR – kehrt hierher zurück
DEB0-   B0 C6       BCS   $DE78     ; „always" – SET SYSERR setzt Carry
```

**– $8x ($81: READ / $82: WRITE) COMMANDS**

Xref: $DE6B – $80/$81 Command Handler Vorstufe

```
DEB2-   A0 05       LDY   #$05
DEB4-   08          PHP             ; Save des INT-Status
DEB5-   78          SEI
DEB6-   B1 40       LDA   ($40),Y   ; Unitnummer, Blocknummer und
DEB8-   99 42 00    STA   $0042,Y     Quell/Zieladresse werden aus dem
DEBB-   88          DEY               „user"-Parameterblock kopiert
DEBC-   D0 F8       BNE   $DEB6
DEBE-   A6 45       LDX   $45       ; Ziel/Quelladresse hi
DEC0-   86 4F       STX   $4F
DEC2-   E8          INX
DEC3-   E8          INX             ; ein Block: 2 Speicherseiten
DEC4-   A5 44       LDA   $44
DEC6-   F0 01       BEQ   $DEC9     ; wenn der Block nicht auf $xx00 anfängt,
DEC8-   E8          INX               dann geht er in eine dritte Seite hinein
DEC9-   20 9F FC    JSR   $FC9F     ; Check mit System Bit Map, s. „Spezielles"
DECC-   B0 08       BCS   $DED6     ; ERR#56: BAD BUFFER ADDRESS, wenn belegt
DECE-   20 DA DE    JSR   $DEDA     ; D0 DISK I/O: RWTB-Aufruf via DRIVER TABLE
DED1-   B0 03       BCS   $DED6     ; bei Fehler: ERR# durch RWTB gesetzt
DED3-   28          PLP
DED4-   18          CLC
DED5-   60          RTS

DED6-   28          PLP             ; ERROR Return
```

Xref: $DEAD – s.o.

```
DED7-   20 09 BF    JSR   $BF09     ; SET SYSERR, für $80/$81 RTS zu $DE78
```

**– DO DISK I/O:** Startadresse der RWTB via Unitnummer aus der DRIVER TABLE

Xref: $DECE – $80/$81 COMMAND Handler
$EC0A – General R/W Block in/aus Directory Space ($DC00) / VBM Space ($DA00)

$EE83 – General R/W Block in/aus einem Filepuffer
$F0E4 – READ, Stufe 3
$F475 – „ACTIVATE" – Test auf „Ready" und WRITE PROTECTED

```
DEDA-  A5 43       LDA   $43          ; Unitnummer
DEDC-  29 F0       AND   #$F0          ; immer DSSS 0000
DEDE-  85 43       STA   $43
DEE0-  4A          LSR
DEE1-  4A          LSR                ; / 8 => 000D SSS0 als Index
DEE2-  4A          LSR
DEE3-  AA          TAX
DEE4-  BD 10 BF    LDA   $BF10,X      ; in der DRIVER TABLE
DEE7-  8D F5 FE    STA   $FEF5
DEEA-  BD 11 BF    LDA   $BF11,X
DEED-  8D F6 FE    STA   $FEF6
DEF0-  6C F5 FE    JMP   ($FEF5)      ; Sprung zur RWTB. RTS von da zum Caller
```

## – $4x ($40: ALLOCATE INT / $41: DEALLOCATE INT) COMMANDS

Xref: $DE57 – Command Dispatcher

```
DEF3-  85 42       STA   $42          ; Aufruf mit 1 ($40) oder 2 ($41) im Akku
DEF5-  4A          LSR
DEF6-  B0 2C       BCS   $DF24        ; $41: DEALLOC INT
DEF8-  A2 03       LDX   #$03         ; ansonsten Start mit erstem Eintrag
DEFA-  BD 7E BF    LDA   $BF7E,X      ; in der INTERRUPT TABLE
DEFD-  D0 16       BNE   $DF15        ; Sprung wenn besetzt
DEFF-  A0 03       LDY   #$03         ; Freier Eintrag gefunden:
DF01-  B1 40       LDA   ($40),Y      ; Startadresse hi aus dem „user"-Programm
DF03-  F0 1A       BEQ   $DF1F        ; = 0  => ERR #53: INVALID PARAMETER
DF05-  9D 7E BF    STA   $BF7E,X
DF08-  88          DEY
DF09-  B1 40       LDA   ($40),Y      ; Startadresse lo
DF0B-  9D 7D BF    STA   $BF7D,X
DF0E-  8A          TXA                ; Index zur 16-Bit-Adresse: Nummer * 2
DF0F-  4A          LSR                ; / 2
DF10-  88          DEY
DF11-  91 40       STA   ($40),Y      ; zurückgelieferte Reference Number
DF13-  18          CLC
DF14-  60          RTS                ; => $DE78

DF15-  E8          INX                ; Eintrag ist besetzt. Nächster Index
DF16-  E8          INX
DF17-  E0 0B       CPX   #$0B         ; Table zu Ende?
DF19-  D0 DF       BNE   $DEFA        ; noch nicht
DF1B-  A9 25       LDA   #$25         ; ERR #25: INTERRUPT TABLE FULL
DF1D-  D0 02       BNE   $DF21
DF1F-  A9 53       LDA   #$53         ; ERR #53: INVALID SYSTEM CALL PARAMETER
DF21-  20 09 BF    JSR   $BF09        ; SET SYSERR: RTS zu $DE78
```

## – $41 COMMAND: DEALLOCATE INTERRUPT

```
DF24-  A0 01       LDY   #$01
DF26-  B1 40       LDA   ($40),Y      ; Reference Number aus dem „user"-Programm
DF28-  F0 F5       BEQ   $DF1F          muß im Bereich 1-4 sein
DF2A-  C9 05       CMP   #$05
DF2C-  B0 F1       BCS   $DF1F        ; sonst ERR #53: INVALID SYSTEM CALL PARM
DF2E-  0A          ASL                ; o.k: Reference Number * 2 als Index
DF2F-  AA          TAX
DF30-  A9 00       LDA   #$00
DF32-  9D 7E BF    STA   $BF7E,X      ; Löschen des Eintrags ohne vorherige
```

```
DF35-  9D 7F BF    STA   $BF7F,X        Prüfung, ob er überhaupt besetzt war
DF38-  18          CLC
DF39-  60          RTS                 ; => $DE78
```

**IRQ HANDLER** ; s. Kapitel „Interrupts der Version 1.1.1"

```
DF3A-  A5 45       LDA   $45           ; „Monitor-IRQ": Akku hier gespeichert
DF3C-  8D 88 BF    STA   $BF88         ; INTAREG
DF3F-  8E 89 BF    STX   $BF89         ; INTXREG
DF42-  8C 8A BF    STY   $BF8A         ; INTYREG
DF45-  BA          TSX
DF46-  8E 8B BF    STX   $BF8B         ; INTSREG
DF49-  AD D8 DF    LDA   $DFD8    *** ; s. „Interrupts der Version 1.1.1"
DF4C-  D0 0C       BNE   $DF5A
DF4E-  68          PLA
DF4F-  8D 8C BF    STA   $BF8C         ; INTPREG - Statusregister
DF52-  68          PLA
DF53-  8D 8E BF    STA   $BF8E
DF56-  68          PLA                 ; INTADDR - Returnadresse
DF57-  8D 8F BF    STA   $BF8F
DF5A-  9A          TXS                 ; Stackpointer wieder auf Anfangsstand
DF5B-  AD F8 07    LDA   $07F8         ; wenn eine Karte „an" ist, ist hier
DF5E-  8D CE DF    STA   $DFCE           die Slotadresse (hi) gespeichert
DF61-  BA          TSX
DF62-  30 09       BMI   $DF6D         ; Sprung wenn Stackpointer > $7F
DF64-  A0 0F       LDY   #$0F
DF66-  68          PLA                 ; sonst werden 16 Werte vom Stack geholt
DF67-  99 DF FE    STA   $FEDF,Y         und gespeichert („Platz machen")
DF6A-  88          DEY                   Der Fehler der Versionen 1.0.1 und 1.0.2
DF6B-  10 F9       BPL   $DF66           (Überschreiben der $C0-$D3 Startadressen)
DF6D-  A2 FA       LDX   #$FA            ist korrigiert worden!
DF6F-  B5 00       LDA   $00,X         ; „Retten" von $FA-$FF auf der Zpage
DF71-  9D F5 FD    STA   $FDF5,X
DF74-  E8          INX
DF75-  D0 F8       BNE   $DF6F

DF77-  AD 81 BF    LDA   $BF81         ; Service 1: Hibyte der Startadresse
DF7A-  F0 05       BEQ   $DF81         ; unbesetzt: Sprung
DF7C-  20 D9 DF    JSR   $DFD9         ; besetzt: Aufruf. SEC wenn nicht zuständig
DF7F-  90 23       BCC   $DFA4
DF81-  AD 83 BF    LDA   $BF83         ; Service 2: Hibyte der Startadresse
DF84-  F0 05       BEQ   $DF8B
DF86-  20 DC DF    JSR   $DFDC         ; besetzt: Aufruf. SEC wenn nicht zuständig
DF89-  90 19       BCC   $DFA4
DF8B-  AD 85 BF    LDA   $BF85         ; Service 3
DF8E-  F0 05       BEQ   $DF95
DF90-  20 DF DF    JSR   $DFDF         ; besetzt: Aufruf. SEC wenn nicht zuständig
DF93-  90 0F       BCC   $DFA4
DF95-  AD 87 BF    LDA   $BF87         ; Service 4
DF98-  F0 05       BEQ   $DF9F
DF9A-  20 E2 DF    JSR   $DFE2         ; besetzt: Aufruf. SEC wenn nicht zuständig
DF9D-  90 05       BCC   $DFA4

DF9F-  A9 01       LDA   #$01          ; keine der Service-Routinen zuständig:
DFA1-  20 0C BF    JSR   $BF0C         ; SYSTEM DEATH mit ERR# 1

DFA4-  A2 FA       LDX   #$FA          ; ok. Restore der Zpage
DFA6-  BD F5 FD    LDA   $FDF5,X
DFA9-  95 00       STA   $00,X
DFAB-  E8          INX
DFAC-  D0 F8       BNE   $DFA6
DFAE-  AE 8B BF    LDX   $BF8B         ; INTSREG: wurde Platz gemacht?
```

```
DFB1-   30 0B       BMI   $DFBE          ; nein, war genug Platz
DFB3-   A0 00       LDY   #$00           ; ansonsten Zurückladen
DFB5-   B9 DF FE    LDA   $FEDF,Y
DFB8-   48          PHA
DFB9-   C8          INY
DFBA-   C0 10       CPY   #$10
DFBC-   D0 F7       BNE   $DFB5
DFBE-   AD D8 DF    LDA   $DFD8      *** ;
DFC1-   D0 12       BNE   $DFD5
DFC3-   AC 8A BF    LDY   $BF8A          ; INTYREG
DFC6-   AE 89 BF    LDX   $BF89          ; INTXREG
DFC9-   AD FF CF    LDA   $CFFF          ; CLRROM - Abschalten aller Karten
DFCC-   AD 00 C1    LDA   $C100          ; REACTIVATE CARD: $DFCE von $DF5E  auf
DFCF-   AD CE DF    LDA   $DFCE            Slotno (hi) gesetzt
DFD2-   8D F8 07    STA   $07F8          ; Restore $07F8
DFD5-   4C D0 BF    JMP   $BFD0          ; INTEXIT in der Global Page
DFD8-   00                           *** ;
```

Xref: Nur vom IRQ Handler via JSR

```
DFD9-   6C 80 BF    JMP   ($BF80)
DFDC-   6C 82 BF    JMP   ($BF82)
DFDF-   6C 84 BF    JMP   ($BF84)
DFE2-   6C 86 BF    JMP   ($BF86)
```

– **SET SYSERR:** Sprung hierher von $BF09

```
DFE5-   8D 0F BF    STA   $BF0F          ; ERR#
DFE8-   68          PLA
DFE9-   68          PLA                  ; eine Returnadrese „rauswerfen"
DFEA-   AD 0F BF    LDA   $BF0F
DFED-   38          SEC                  ; Indikator: ERROR
DFEE-   60          RTS                  ; Exit normalerweise zu $DE78
```

– **SYSTEM DEATH:** Entry Nummer 2: Setzt ERR# = 00

```
DFEF-   A9 00       LDA   #$00
```

– **SYSTEM DEATH:** Entry Nummer 1: Aufruf mit ERR# in A, Sprung hierher von $BF0C

```
DFF1-   AA          TAX                  ; ERR#
DFF2-   8D 0C C0    STA   $C00C          ; Abschalten der 80-Zeichen-Karte (IIc/e)
DFF5-   AD 51 C0    LDA   $C051          ; TEXT
DFF8-   AD 53 C0    LDA   $C053          ; MIX
DFFB-   AD 54 C0    LDA   $C054          ; PAGE 1
DFFE-   AD 56 C0    LDA   $C056          ; LORES
E001-   A0 27       LDY   #$27
E003-   A9 A0       LDA   #$A0           ; Space
E005-   99 50 07    STA   $0750,Y
E008-   B9 1E FE    LDA   $FE1E,Y        ; „INSERT SYSTEM DISK AND RESTART"
E00B-   99 D0 07    STA   $07D0,Y        ; direkt in den Bildspeicher
E00E-   88          DEY
E00F-   10 F2       BPL   $E003
E011-   8A          TXA                  ; ERR#
E012-   F0 30       BEQ   $E044          ; 00: „-ERRxx" wird nicht gedruckt
E014-   A9 AD       LDA   #$AD
E016-   8D F1 07    STA   $07F1          ; „-"
E019-   A9 C5       LDA   #$C5
E01B-   8D F2 07    STA   $07F2          ; „E"
E01E-   A9 D2       LDA   #$D2
E020-   8D F3 07    STA   $07F3          ; „R"
```

```
E023-  8D F4 07    STA   $07F4          ; „R"
E026-  8A          TXA                  ; ERR#
E027-  4A          LSR
E028-  4A          LSR
E029-  4A          LSR
E02A-  4A          LSR                  ; Übersetzung des höheren Nibble
E02B-  09 B0       ORA   #$B0
E02D-  C9 BA       CMP   #$BA
E02F-  90 02       BCC   $E033
E031-  69 06       ADC   #$06           ; auf „A"-„F" für Werte > 9
E033-  8D F6 07    STA   $07F6          ; „Print"
E036-  8A          TXA
E037-  29 0F       AND   #$0F           ; niederwertiges Nibble der Hexzahl
E039-  09 B0       ORA   #$B0
E03B-  C9 BA       CMP   #$BA
E03D-  90 02       BCC   $E041
E03F-  69 06       ADC   #$06           ; „A"-„F" für Werte > 9
E041-  8D F7 07    STA   $07F7          ; „Print"

E044-  4C 44 E0    JMP   $E044          ; bis zum nächsten RESET
```

# $Cx, $Dx COMMANDS

Xref: $DE75 – Command Dispatcher

```
E047-  8E B7 FE    STX   $FEB7          ; Aufruf mit Index im Bereich $00-$13
E04A-  BD CD FD    LDA   $FDCD,X        ; Bit-Table: Bit 7-5 bestimmen, welche
E04D-  0A          ASL                    Routinen vorher aufgerufen werden
E04E-  8D B3 FE    STA   $FEB3
E051-  29 3F       AND   #$3F           ; Bit 0-4 sind der Index (* 2) für die
E053-  AA          TAX                    Startadresse des eigentlichen
E054-  BD A5 FD    LDA   $FDA5,X        ; Kommandos in der nächsten Tabelle
E057-  8D F5 FE    STA   $FEF5
E05A-  BD A6 FD    LDA   $FDA6,X        ; $FEF5/F6: Startadresse des Commands
E05D-  8D F6 FE    STA   $FEF6
E060-  A9 20       LDA   #$20
E062-  8D B4 FE    STA   $FEB4          ; internes BACKUPBIT – wird immer gesetzt
E065-  90 05       BCC   $E06C          ; Carry vom ersten „ASL" ($E04D)
E067-  20 8A E0    JSR   $E08A          ; „Copy & Verify Pathname"
E06A-  B0 17       BCS   $E083          ; Error: => SET SYSERR => $DE78
E06C-  0E B3 FE    ASL   $FEB3
E06F-  90 05       BCC   $E076
E071-  20 D0 E1    JSR   $E1D0          ; „Set File Data"
E074-  B0 0D       BCS   $E083          ; Error: => SET SYSERR => $DE78
E076-  0E B3 FE    ASL   $FEB3
E079-  90 03       BCC   $E07E
E07B-  20 06 BF    JSR   $BF06          ; SET TIME – keine Fehler möglich
E07E-  20 87 E0    JSR   $E087          ; Ausführung des Command
E081-  90 03       BCC   $E086          ; keine Fehler: RTS zu $DE78
E083-  20 09 BF    JSR   $BF09          ; SET SYSERR: RTS zu $DE78
E086-  60          RTS

E087-  6C F5 FE    JMP   ($FEF5)        ; ersetzt „JSR (xx)"
```

### – COPY & VERIFY PATHNAME

Aufruf vor CREATE/DESTROY/RENAME/SET FILE INFO/GET FILE INFO/OPEN
Der Pathname wird aus dem Parameterblock des „user"-Programms geholt, nach $F700... kopiert und verifiziert. Dabei wird auf Großbuchstaben übersetzt und auf illegale Zeichen

geprüft. Das erste Zeichen jedes Namens muß ein Buchstabe sein, die Länge des gesamten Namens darf $40 nicht überschreiten.
Der Path erhält dabei die folgende Form:
Länge „NAME1" Länge „NAME2" ... Länge „NAMEn" 00 <Ende>

Xref: $E067 – $C0-$D3 Commands
$E135 – SET PREFIX (interner Aufruf – ERR#40 für PREFIX „/" wird abgefangen)
$EAB2 – RENAME (vor der Suche des „alten" Files)

```
E08A-  A0 01     LDY  #$01
E08C-  B1 40     LDA  ($40),Y      ; Pointer zum  Pathname aus dem „user"-
E08E-  85 48     STA  $48            Programm auf $0048/49
E090-  C8        INY
E091-  B1 40     LDA  ($40),Y
E093-  85 49     STA  $49
```

Xref: JP $F956 – RENAME: $0048/49 auf den „neuen" Path gesetzt

```
E095-  A2 00     LDX  #$00
E097-  A0 00     LDY  #$00
E099-  8E BC FE  STX  $FEBC        ; Reset FULLNAME FLAG
E09C-  8E 00 D7  STX  $D700        ; Länge des Pathname jetzt: Null
E09F-  B1 48     LDA  ($48),Y      ; Länge des gesamten Pathname
E0A1-  F0 58     BEQ  $E0FB        ; 0: ERR #40: INVALID PATHNAME
E0A3-  C9 41     CMP  #$41
E0A5-  B0 54     BCS  $E0FB        ; >$40: ERR #40: INVALID PATHNAME
E0A7-  8D 9E FE  STA  $FE9E
E0AA-  EE 9E FE  INC  $FE9E        ; max. Index
E0AD-  C8        INY
E0AE-  B1 48     LDA  ($48),Y
E0B0-  09 80     ORA  #$80
E0B2-  C9 AF     CMP  #$AF         ; Start mit „/"?
E0B4-  D0 04     BNE  $E0BA
E0B6-  8D BC FE  STA  $FEBC        ; wenn ja, set FULLNAME FLAG
E0B9-  C8        INY               ; das „/" wird in diesem Fall übersprungen

E0BA-  A9 FF     LDA  #$FF         ; Indikator: erstes Zeichen im Namen
E0BC-  9D 00 D7  STA  $D700,X
E0BF-  8D B8 FE  STA  $FEB8        ; und Längenzähler für einen Namen im Path
E0C2-  8E BA FE  STX  $FEBA        ; Start-Index für diesen Namen im Path
E0C5-  CC 9E FE  CPY  $FE9E        ; Ende des gesamten Path?
E0C8-  B0 35     BCS  $E0FF
E0CA-  B1 48     LDA  ($48),Y      ; ein Zeichen
E0CC-  29 7F     AND  #$7F         ; Bit 7 wird maskiert
E0CE-  E8        INX
E0CF-  C8        INY
E0D0-  C9 2F     CMP  #$2F         ; „/"?
E0D2-  F0 40     BEQ  $E114        ; wenn ja, Ende des Namens (nicht des Path!)
E0D4-  C9 61     CMP  #$61
E0D6-  90 02     BCC  $E0DA
E0D8-  29 5F     AND  #$5F         ; wenn >„a", Übersetzung auf Großbuchstaben
E0DA-  9D 00 D7  STA  $D700,X
E0DD-  EE B8 FE  INC  $FEB8        ; Start mit $FF als Flag für „erstes Zeichen"
E0E0-  D0 05     BNE  $E0E7
E0E2-  EE B8 FE  INC  $FEB8        ; wenn dies das erste Zeichen im Namen war
E0E5-  D0 0C     BNE  $E0F3          Erhöhung um 2 und nur Test auf Buchstabe
E0E7-  C9 2E     CMP  #$2E
E0E9-  F0 DA     BEQ  $E0C5        ; „." ist ok: NEXT
E0EB-  C9 30     CMP  #$30
```

```
E0ED-  90 0C     BCC  $E0FB       ; < „0": INVALID PATHNAME
E0EF-  C9 3A     CMP  #$3A
E0F1-  90 D2     BCC  $E0C5       ; „0"-„9" ist ok: NEXT
E0F3-  C9 41     CMP  #$41
E0F5-  90 04     BCC  $E0FB       ; > „9" and < „A": INVALID PATHNAME
E0F7-  C9 5B     CMP  #$5B
E0F9-  90 CA     BCC  $E0C5       ; „A"-„Z" ist ok: NEXT

E0FB-  38        SEC              ; ERR #40: INVALID PATHNAME
E0FC-  A9 40     LDA  #$40
E0FE-  60        RTS

E0FF-  A9 00     LDA  #$00        ; Ende eines Namens im Path
E101-  2C B8 FE  BIT  $FEB8       ; Länge dieses Namens: $FF, wenn der
E104-  10 04     BPL  $E10A         Name nur aus „/" besteht => Längenbyte
E106-  8D B8 FE  STA  $FEB8         auf $00 (kein Fehler!)
E109-  CA        DEX              ** ; auf $FF wenn der GESAMTE Name nur
E10A-  E8        INX              **   aus „/" besteht
E10B-  9D 00 D7  STA  $D700,X     ** ; $00 nach dem Namen
E10E-  F0 EB     BEQ  $E0FB       ** ; Z-flag von INX! : ERR #40
E110-  8E 9E FE  STX  $FE9E       ** ; Gesamtlänge des Pathname
E113-  AA        TAX              ## ; = $00 !

E114-  AD B8 FE  LDA  $FEB8       ** ; Einsprung, wenn „/" gefunden: Länge
E117-  C9 10     CMP  #$10
E119-  B0 E0     BCS  $E0FB       ** ; dieser Name ist länger als $0F Zeichen
E11B-  8E BD FE  STX  $FEBD       ## ; Save des momentanen Index
E11E-  AE BA FE  LDX  $FEBA       ** ; Reload Startindex dieses Namens im Path
E121-  9D 00 D7  STA  $D700,X     ** ; Speichern der Länge als erstes Byte
E124-  AE BD FE  LDX  $FEBD       ## ; Reload momentanen Index
E127-  D0 91     BNE  $E0BA       ## ; wenn von $D2F4: Ende
E129-  18        CLC
E12A-  AD BC FE  LDA  $FEBC       ** ; FULLNAME FLAG: vollständiger Name?
E12D-  D0 05     BNE  $E134       ** ; Ja. Kein System Prefix dazu
E12F-  AD 9A BF  LDA  $BF9A       ** ; ansonsten check PREFIX
E132-  F0 C7     BEQ  $E0FB       ** ; kein PREFIX => ERR #40
E134-  60        RTS
```

## $C6 – SET PREFIX COMMAND ; keine Routinen vorher aufgerufen

```
E135-  20 8A E0  JSR  $E08A       ** ; Copy & Verify Pathname
E138-  90 0A     BCC  $E144       ** ; Länge des ersten Namens im Path
E13A-  AC 00 D7  LDY  $D700       ** ; <> 0 und ERR => EXIT mit ERR#
E13D-  D0 4C     BNE  $E18B       ** ; ist 0: Reset System PREFIX
E13F-  8C 9A BF  STY  $BF9A         das ist ein gültiges Command!
E142-  18        CLC
E143-  60        RTS

E144-  20 A3 E5  JSR  $E5A3       ** ; normaler Pathname: GET FILE
E147-  90 04     BCC  $E14D
E149-  C9 40     CMP  #$40        ** ; INVALID (Ende in VolDIR) ist kein Fehler
E14B-  D0 3E     BNE  $E18B       ** ; ansonsten Exit mit ERR#
E14D-  AD 5F FE  LDA  $FE5F       ** ; Storage Type des Files –
E150-  29 D0     AND  #$D0          muß eine (Sub)directory sein
E152-  49 D0     EOR  #$D0
E154-  D0 33     BNE  $E189       ** ; ERR #4B: UNSUPPORTED STORAGE TYPE
E156-  AC BC FE  LDY  $FEBC       ** ; FULLNAME FLAG
E159-  D0 03     BNE  $E15E
E15B-  AD 9A BF  LDA  $BF9A       ** ; sonst Start mit Index für SYS PREFIX
E15E-  A8        TAY              ** ; 0 für „Full Name", sonst Index von PREFIX
E15F-  38        SEC
E160-  ED 9E FE  SBC  $FE9E       ** ; minus Länge des neuen Namens
```

```
E163-  C9 C0     CMP   #$C0          ; alles zusammen länger als $40 Zeichen?
E165-  90 94     BCC   $E0FB          ; Exit mit ERR #40
E167-  AA        TAX
E168-  8D 9A BF  STA   $BF9A          ; sonst Update von PREFIX zum neuen Start
E16B-  AD 59 FE  LDA   $FE59          ; Unitnummer des PREFIX-Units
E16E-  8D 9F FE  STA   $FE9F          ; (nie mehr benutzt)
E171-  AD 70 FE  LDA   $FE70          ; Nummer des Key Blocks der PREFIX-Dir
E174-  8D A0 FE  STA   $FEA0
E177-  AD 71 FE  LDA   $FE71          ; PREFIX DIRECTORY
E17A-  8D A1 FE  STA   $FEA1
E17D-  B9 00 D7  LDA   $D700,Y        ; 0, wenn FULLNAME, ansonsten Start mit
                                        altem PREFIX und OVERFLOW in den neuen
E180-  9D 00 D7  STA   $D700,X          Namen, d.h. Anhängen des neuen Namens
E183-  C8        INY
E184-  E8        INX                  ; bis X (nicht Y!) = 0
E185-  D0 F6     BNE   $E17D          ; der neue (Gesamt)name fängt im Bereich
E187-  18        CLC                    $F7C0..$F7FE an
E188-  60        RTS                  ; => $DE78

E189-  A9 4B     LDA   #$4B           ; ERR #4B: UNSUPPORTED STORAGE TYPE
E18B-  38        SEC
E18C-  60        RTS
```

## $C7 – GET PREFIX COMMAND ; keine anderen Routinen vorher

```
E18D-  18        CLC
E18E-  A0 01     LDY   #$01           ; Startadresse des „user"-Puffers
E190-  B1 40     LDA   ($40),Y          wird auf $004E/4F kopiert
E192-  85 4E     STA   $4E
E194-  C8        INY
E195-  B1 40     LDA   ($40),Y
E197-  85 4F     STA   $4F
E199-  A9 00     LDA   #$00
E19B-  8D DB FE  STA   $FEDB
E19E-  A9 40     LDA   #$40           ; Platzbedarf: $40 Zeichen (max.)
E1A0-  8D DA FE  STA   $FEDA
E1A3-  20 82 FC  JSR   $FC82          ; Check mit System Bit Map
E1A6-  B0 E3     BCS   $E18B          ; ERR #56: BAD BUFFER ADDRESS
E1A8-  A0 00     LDY   #$00
E1AA-  AD 9A BF  LDA   $BF9A          ; PREFIX
E1AD-  AA        TAX
E1AE-  F0 04     BEQ   $E1B4          ; $00: kein Prefix (gültig!)
E1B0-  49 FF     EOR   #$FF
E1B2-  69 02     ADC   #$02           ; Längenberechung vom Startindex ($F1C0-FF)
E1B4-  91 4E     STA   ($4E),Y        ; erstes Byte im „user"-Puffer
E1B6-  F0 16     BEQ   $E1CE          ; wenn $00 => Kopie beendet
E1B8-  C8        INY
E1B9-  BD 00 D7  LDA   $D700,X
E1BC-  91 4E     STA   ($4E),Y        ; Kopie in den „user"-Puffer
E1BE-  29 F0     AND   #$F0
E1C0-  D0 04     BNE   $E1C6          ; wenn dabei $00 herauskommt, ist das ein
E1C2-  A9 2F     LDA   #$2F             Längenbyte ($00-$0F) und wird durch „/"
E1C4-  D0 F6     BNE   $E1BC            ersetzt
E1C6-  E8        INX
E1C7-  D0 EF     BNE   $E1B8          ; Loop bis Index = $00
E1C9-  C8        INY
E1CA-  A9 2F     LDA   #$2F           ; ein letzter „/" wird angehängt
E1CC-  91 4E     STA   ($4E),Y
E1CE-  18        CLC
E1CF-  60        RTS                  ; => $DE78
```

**– SET FILE DATA**

Aufruf vor NEWLINE/READ/WRITE/SET & GET: MARK/EOF/BUF
Überprüfung der Reference Number und ob der Eintrag im FCB existiert.
Setzt Filepuffer-Adresse, VCB-Index und berechnet den FCB-Index.
Existiert kein VCB-Eintrag/kein Filepuffer für einen OPEN File, folgt SYSTEM DEATH.

Xref: $E071 – $C0-$D3 Commands
$F52B – FLUSH (single file – das Argument $00 (ALL) wird vorher abgefangen)

```
E1D0-   A0 01       LDY   #$01
E1D2-   B1 40       LDA   ($40),Y       ; Reference Number vom „user"
E1D4-   F0 5B       BEQ   $E231
E1D6-   C9 09       CMP   #$09          ; muß im Bereich 1-8 sein
E1D8-   B0 57       BCS   $E231         ; ERR#43: INVALID REFERENCE NUMBER
E1DA-   48          PHA
E1DB-   E9 00       SBC   #$00          ; minus 1 => 0-7
E1DD-   4A          LSR
E1DE-   6A          ROR
E1DF-   6A          ROR
E1E0-   6A          ROR
E1E1-   8D 92 FE    STA   $FE92         ; * $20 => FCB-Index
E1E4-   A8          TAY
E1E5-   68          PLA
E1E6-   D9 00 D8    CMP   $D800,Y       ; der Eintrag muß die Reference Number
E1E9-   D0 41       BNE   $E22C           enthalten, d.h. existieren
```

– Laden der FCB-Parameter und Setzen des VCB-Index

Xref: $EF59 – OPEN (Setzen der Parameter für das Lesen des ersten Blocks – SET MARK)
JP $F1B8 – READ (Re-set der Filepuffer-Pointer nach Stufe 2/3)
$F51E – FLUSH (ein File im FLUSH ALL-Loop)

```
E1EB-   B9 0B D8    LDA   $D80B,Y       ; Puffernummer * 2
E1EE-   20 3C FC    JSR   $FC3C         ; Pufferstart von SGP => $FEDD/DE
E1F1-   AE DE FE    LDX   $FEDE         ; kein Puffer (hi-Adresse = $00)?
E1F4-   F0 27       BEQ   GP => $FEDD/DE
E1F1-   AE DE FE    LDX   $FEDE         ; kein Puffer (hi-Adresse = $00)?
E1F4-   F0 27       BEQ   $E21D         ; SYSTEM DEATH #0B
E1F6-   86 4B       STX   $4B           ; Start des Datenteils
E1F8-   E8          INX
E1F9-   E8          INX
E1FA-   86 49       STX   $49           ; Indexteil ($200 darüber)
E1FC-   B9 01 D8    LDA   $D801,Y       ; Unitnummer
E1FF-   8D 30 BF    STA   $BF30
E202-   AD DD FE    LDA   $FEDD
E205-   85 4A       STA   $4A           ; Pufferstarts lo (immer 00)
E207-   85 48       STA   $48
E209-   AA          TAX
E20A-   BD 10 D9    LDA   $D910,X       ; Suche nach dem VCB-Eintrag, d.h. dem
E20D-   D9 01 D8    CMP   $D801,Y         Unit, auf dem der File existiert
E210-   F0 10       BEQ   $E222         ; gefunden
E212-   8A          TXA
E213-   18          CLC
E214-   69 20       ADC   #$20          ; nächster VCB-Eintrag
E216-   90 F1       BCC   $E209
```

```
E218-  A9 0A       LDA   #$0A          ; KEIN UNIT FÜR DEN FILE!
E21A-  20 0C BF    JSR   $BF0C          SYSTEM DEATH #0A
E21D-  A9 0B       LDA   #$0B          ; KEIN BUFFER FÜR DEN FILE!
E21F-  20 0C BF    JSR   $BF0C          SYSTEM DEATH #0B

E222-  BD 00 D9    LDA   $D900,X       ; VCB-Eintrag mit passender Unitno gefunden
E225-  F0 EB       BEQ   $E212         ; wenn ungültig, nächster Eintrag
E227-  8E 91 FE    STX   $FE91         ; VCB-Index
E22A-  18          CLC
E22B-  60          RTS                 ; zur aufrufenden Routine

E22C-  A9 00       LDA   #$00          ; ERR Exit: kein FCB-Eintrag
E22E-  99 92 FE    STA   $FE92,Y  **** ; Programmierfehler, s. „Spezielles"
E231-  A9 43       LDA   #$43          ; ERR#43: INVALID REFERENCE NUMBER
E233-  38          SEC
E234-  60          RTS                 ; zur aufrufenden Routine
```

**$C5 – ONLINE COMMAND** ; keine anderen Routinen vorher

```
E235-  20 0D F2    JSR   $F20D         ; Startadresse des „user"-Puffers =>  $004E/4F
E238-  A9 00       LDA   #$00
E23A-  8D DA FE    STA   $FEDA
E23D-  8D DB FE    STA   $FEDB         ; Annahme: benötigter Platz = 00 00
E240-  A0 01       LDY   #$01
E242-  B1 40       LDA   ($40),Y       ; Reference number
E244-  F0 07       BEQ   $E24D         ; $00 = „ALL"
E246-  A9 10       LDA   #$10
E248-  8D DA FE    STA   $FEDA         ; Spezifische Refno: $10 Bytes Platz
E24B-  D0 05       BNE   $E252
E24D-  A9 01       LDA   #$01          ; „ALL": $100 Bytes Platz werden gebraucht
E24F-  8D DB FE    STA   $FEDB
E252-  20 82 FC    JSR   $FC82         ; Check mit System Bit Map
E255-  B0 33       BCS   $E28A         ; ERR #56: BAD BUFFER ADDRESS
E257-  A9 00       LDA   #$00
E259-  AC DA FE    LDY   $FEDA
E25C-  88          DEY
E25D-  91 4E       STA   ($4E),Y       ; der Puffer ($10 oder $100 Bytes)
E25F-  D0 FB       BNE   $E25C           wird gelöscht
E261-  8D BA FE    STA   $FEBA         ; erster Index im Puffer: $00
E264-  A0 01       LDY   #$01
E266-  B1 40       LDA   ($40),Y       ; Refno
E268-  29 F0       AND   #$F0
E26A-  D0 1F       BNE   $E28B         ; Spezifisch: Sprung zu ONLINE SPECIFIC
E26C-  20 64 E8    JSR   $E864         ; „ALL": DEVTBL wird auf $FECA... kopiert

E26F-  8E BD FE    STX   $FEBD         ; DEVNUM: Anzahl der DEVICES
E272-  BD CA FE    LDA   $FECA,X
E275-  20 8B E2    JSR   $E28B         ; Für jedes DEVICE ein ONLINE SPECIFIC
E278-  AD BA FE    LDA   $FEBA
E27B-  18          CLC
E27C-  69 10       ADC   #$10          ; nächster Index im Puffer
E27E-  8D BA FE    STA   $FEBA
E281-  AE BD FE    LDX   $FEBD         ; nächstes DEVICE
E284-  CA          DEX
E285-  10 E8       BPL   $E26F
E287-  A9 00       LDA   #$00
E289-  18          CLC
E28A-  60          RTS                 ; => $DE78
```

**– ONLINE SPECIFIC:** ONLINE für ein Unit

Xref: $E26A – ONLINE COMMAND mit spezifizierter Unitnummer
$E275 – ONLINE „ALL" Loop

```
E28B-  8D 30 BF    STA   $BF30        ; Aufruf mit Unitnummer im Akku
E28E-  20 76 E8    JSR   $E876        ; sucht nach diesem Unit im VCB oder nach
E291-  B0 30       BCS   $E2C3          einem freien Eintrag. ERR wenn VCB voll
E293-  A9 00       LDA   #$00           VCB-Index ($F051) ist auf gefundenen oder
E295-  85 47       STA   $47            freien Eintrag gesetzt
E297-  A9 02       LDA   #$02
E299-  85 46       STA   $46
E29B-  20 EE EB    JSR   $EBEE        ; der Key Block der VolDIR wird gelesen
E29E-  B0 23       BCS   $E2C3        ; ERR - kein Exit!
E2A0-  AE 91 FE    LDX   $FE91
E2A3-  BD 00 D9    LDA   $D900,X
E2A6-  F0 05       BEQ   $E2AD        ; der VCB-Eintrag war frei, Sprung
E2A8-  BD 11 D9    LDA   $D911,X      ; nicht frei: OPEN File?
E2AB-  30 0C       BMI   $E2B9        ; wenn ja, Vergleich neuer/alter Vol-Name

E2AD-  20 D1 E8    JSR   $E8D1        ; VCB-Eintrag wird mit neuer VolDIR besetzt
E2B0-  B0 11       BCS   $E2C3        ; ERR: kein ProDOS-Volume. Kein Exit!
E2B2-  A9 57       LDA   #$57         ; ERR#57: DUPLICATE VOLUME (kein Exit),
E2B4-  2C B5 FE    BIT   $FEB5          wenn $E876 ein Volume mit OPEN File und
E2B7-  30 0A       BMI   $E2C3          gleichem Namen gefunden hat ($F0B5 = $80)

E2B9-  AE 91 FE    LDX   $FE91        ; Vergleich alter VCB-Eintrag/neue VolDIR:
E2BC-  20 29 E9    JSR   $E929          Nur sinnvoll, wenn Sprung von $E2AB
E2BF-  A9 2E       LDA   #$2E         ; ERR#2E: DISK SWITCHED AND FILE OPEN
E2C1-  90 1E       BCC   $E2E1        ; Sprung, wenn ok. ERR# ist dann ungültig
```

– ERR Handler für ONLINE

```
E2C3-  48          PHA                ; ERR#
E2C4-  20 F6 E2    JSR   $E2F6        ; Unitnummer in den „user"-Puffer
E2C7-  68          PLA
E2C8-  C8          INY
E2C9-  91 4E       STA   ($4E),Y      ; und ERR# danach
E2CB-  C9 57       CMP   #$57         ; war ERR#: DUPLICATE VOLUME?
E2CD-  D0 10       BNE   $E2DF
E2CF-  C8          INY
E2D0-  AE B6 FE    LDX   $FEB6        ; Unitnummer des doppelten Vols, von $E876
E2D3-  BD 10 D9    LDA   $D910,X
E2D6-  91 4E       STA   ($4E),Y      ; als zweites Byte in den „user"-Puffer
E2D8-  A9 00       LDA   #$00
E2DA-  8D B5 FE    STA   $FEB5        ; Reset des OPEN File-Flags
E2DD-  A9 57       LDA   #$57
E2DF-  38          SEC
E2E0-  60          RTS                ; => $DE78 oder ONLINE ALL-Loop
```

–keine Fehler in ONLINE SPECIFIC

```
E2E1-  BD 00 D9    LDA   $D900,X      ; Länge des Volume-Namens
E2E4-  8D B8 FE    STA   $FEB8        ; Zähler
E2E7-  AC BA FE    LDY   $FEBA        ; Start-Index im „user"-Puffer
E2EA-  BD 00 D9    LDA   $D900,X
E2ED-  91 4E       STA   ($4E),Y      ; der Name wird kopiert
E2EF-  E8          INX
E2F0-  C8          INY
E2F1-  CE B8 FE    DEC   $FEB8
E2F4-  10 F4       BPL   $E2EA
```

– Fügt die Unitnummer zur Namenslänge hinzu

Xref: $E2C4 – ONLINE Error Handler
$E2F4 – ONLINE ohne Fehler

```
E2F6-  AC BA FE    LDY   $FEBA          ; Start-Index im „user"-Puffer
E2F9-  AD 30 BF    LDA   $BF30
E2FC-  11 4E       ORA   ($4E),Y        ; das Längenbyte des Namens ($00-$0F)
E2FE-  91 4E       STA   ($4E),Y          bekommt die Unitnummer (DSSS 0000) dazu
E300-  18          CLC
E301-  60          RTS                  ; => ONLINE ERROR/$DE78/ONLINE „ALL" Loop
```

## $C0 – CREATE COMMAND

Xref: $E3B1 – CREATE selber nach Anhängen eines Subdirectory-Blocks

```
E302-  20 B6 E5    JSR   $E5B6          ; FIND FILE ENTRY
E305-  B0 04       BCS   $E30B          ; muß FILE NOT FOUND ergeben!
E307-  A9 47       LDA   #$47           ; ERR#47: DUPLICATE FILENAME
E309-  38          SEC
E30A-  60          RTS                  ; => $DE78
E30B-  C9 46       CMP   #$46           ; FILE NOT FOUND?
E30D-  D0 FA       BNE   $E309          ; andere Fehler werden nicht maskiert
E30F-  A0 07       LDY   #$07
E311-  B1 40       LDA   ($40),Y        ; Storage Type vom „user"
E313-  C9 04       CMP   #$04
E315-  90 04       BCC   $E31B          ; „normaler" File
E317-  C9 0D       CMP   #$0D           ; Subdirectory?
E319-  D0 10       BNE   $E32B          ; ERR#4E: UNSUPPORTED STORAGE TYPE
E31B-  AD 30 BF    LDA   $BF30          ; Unitnummer
E31E-  20 65 F4    JSR   $F465          ; Test auf WRITE PROTECTED
E321-  B0 0B       BCS   $E32E
E323-  AD 9B FE    LDA   $FE9B          ; von FIND FILE ENTRY: Nummer eines freien
E326-  F0 07       BEQ   $E32F            Dir-Eintrags. $00 = kein freier Eintrag
E328-  4C C1 E3    JMP   $E3C1          ; freier Eintrag vorhanden

E32B-  A9 4B       LDA   #$4B           ; ERR#4B: UNSUPPORTED STORAGE TYPE
E32D-  38          SEC
E32E-  60          RTS                  ; => $DE78
```

## – APPEND A BLOCK TO SUBDIRECTORY

– kein freier Eintrag von FIND FILE ENTRY gefunden. Danach Restart von CREATE

```
E32F-  AD 46 FE    LDA   $FE46          ; Parent Pointer der momentanen DIR
E332-  0D 47 FE    ORA   $FE47            Eine Volume Directory hat den Pointer
E335-  D0 04       BNE   $E33B            auf 00 00 gesetzt
E337-  A9 49       LDA   #$49           ; ERR#49: VOLUME DIRECTORY FULL
E339-  38          SEC
E33A-  60          RTS                  ; => $DE78
E33B-  A5 46       LDA   $46            ; Nummer des momentanen (letzten) DirBlocks
E33D-  48          PHA
E33E-  A5 47       LDA   $47
E340-  48          PHA
E341-  20 B6 EA    JSR   $EAB6          ; ALLOCATE NEXT BLOCK
E344-  AA          TAX                  ; eventuelle ERR#, ansonsten Blocknummer
E345-  68          PLA                    des neuen Blocks (low)
E346-  85 47       STA   $47
E348-  68          PLA
E349-  85 46       STA   $46
E34B-  8A          TXA
E34C-  B0 E0       BCS   $E32E          ; ERR# während ALLOCATE NEXT BLOCK
E34E-  8D 02 DC    STA   $DC02          ; die Nummer des neuen Blocks wird im
E351-  8C 03 DC    STY   $DC03            „alten" Block als Forward Ref gesetzt
E354-  20 EA EB    JSR   $EBEA            der DirBlock wird zurückgeschrieben
E357-  B0 D5       BCS   $E32E
E359-  A2 01       LDX   #$01
```

```
E35B-  B5 46      LDA  $46,X       ; die Nummer des „alten" DirBlocks wird im
E35D-  9D 00 DC   STA  $DC00,X       „neuen" Block als Backward Ref gesetzt
E360-  BD 02 DC   LDA  $DC02,X
E363-  95 46      STA  $46,X       ; und die Nummer des „neuen" Blocks
E365-  CA         DEX                (Forward Ref im „alten" Block) als
E366-  10 F3      BPL  $E35B         Parameter für die RWTB
E368-  E8         INX
E369-  8A         TXA
E36A-  9D 02 DC   STA  $DC02,X     ; der gesamte „neue" Block wird bis auf
E36D-  9D 00 DD   STA  $DD00,X       die Backward Reference gelöscht
E370-  E8         INX
E371-  D0 F7      BNE  $E36A
E373-  20 EA EB   JSR  $EBEA       ; und auf das Volume geschrieben
E376-  B0 B6      BCS  $E32E
E378-  AD 46 FE   LDA  $FE46       ; Nummer des Parent Blocks
E37B-  85 46      STA  $46
E37D-  AD 47 FE   LDA  $FE47
E380-  85 47      STA  $47         ; der Parent Block der Subdirectory
E382-  20 EE EB   JSR  $EBEE       ; wird gelesen
E385-  AE 48 FE   LDX  $FE48       ; Eintragsnummer in diesem Block
E388-  A9 DC      LDA  #$DC
E38A-  85 49      STA  $49
E38C-  A9 04      LDA  #$04
E38E-  18         CLC              ; $0048/49: Startadresse des Eintrags
E38F-  CA         DEX
E390-  F0 09      BEQ  $E39B
E392-  6D 49 FE   ADC  $FE49
E395-  90 F7      BCC  $E38E
E397-  E6 49      INC  $49
E399-  B0 F3      BCS  $E38E

E39B-  85 48      STA  $48
E39D-  A0 13      LDY  #$13        ; BLOCKS USED und EOF werden auf den
E39F-  B1 48      LDA  ($48),Y       neuen Stand gebracht:
E3A1-  79 CE FD   ADC  $FDCE,Y       BLOCKS USED plus 1
E3A4-  91 48      STA  ($48),Y       EOF plus $200
E3A6-  C8         INY
E3A7-  98         TYA
E3A8-  49 18      EOR  #$18        ; anstelle von CMP (Carry!)
E3AA-  D0 F3      BNE  $E39F
E3AC-  20 EA EB   JSR  $EBEA       ; der Parent Block wird zurückgeschrieben
E3AF-  B0 0F      BCS  $E3C0
E3B1-  4C 02 E3   JMP  $E302       ; Kompletter Restart von CREATE
```

– löscht den gesamten Directory Space

Xref: $E3C1 – CREATE Subdirectory Key Block oder eines leeren Datenblocks

```
E3B4-  A9 00      LDA  #$00
E3B6-  AA         TAX
E3B7-  9D 00 DC   STA  $DC00,X
E3BA-  9D 00 DD   STA  $DD00,X
E3BD-  E8         INX
E3BE-  D0 F7      BNE  $E3B7
E3C0-  60         RTS
```

– Erstellung eines Subdirectory Key oder eines leeren Datenblocks
– Erstellung eines File Info für den Eintrag in die Directory

Xref: $E328 – JP von CREATE, wenn ein freier Directory-Eintrag gefunden wurde

```
E3C1-  20 B4 E3    JSR   $E3B4          ; löscht den Directory Space
E3C4-  A0 0B       LDY   #$0B
E3C6-  B1 40       LDA   ($40),Y        ; CREATE DATE & TIME vom „user"
E3C8-  99 6F FE    STA   $FE6F,Y
E3CB-  8A          TXA
E3CC-  11 40       ORA   ($40),Y
E3CE-  AA          TAX                  ; ORA für alle vier Bytes
E3CF-  88          DEY
E3D0-  C0 07       CPY   #$07
E3D2-  D0 F2       BNE   $E3C6
E3D4-  8A          TXA                  ; Resultat des ORA
E3D5-  D0 0B       BNE   $E3E2          ; DATE & TIME war vom „user" spezifiziert
E3D7-  A2 03       LDX   #$03
E3D9-  BD 90 BF    LDA   $BF90,X        ; ansonsten wird die System Time kopiert
E3DC-  9D 77 FE    STA   $FE77,X
E3DF-  CA          DEX
E3E0-  10 F7       BPL   $E3D9
E3E2-  B1 40       LDA   ($40),Y        ; Storage Type vom „user"
E3E4-  C9 04       CMP   #$04
E3E6-  A9 10       LDA   #$10           ; 0(!)/1/2/3 => $10
E3E8-  90 02       BCC   $E3EC
E3EA-  A9 D0       LDA   #$D0           ; $D0: Subdirectory
E3EC-  AE BA FE    LDX   $FEBA          ; Start-Index des letzten Namens im Path
E3EF-  1D 00 D7    ORA   $D700,X          (von FIND FILE ENTRY übriggeblieben)
E3F2-  8D 5F FE    STA   $FE5F          ; = Storage Type/Namenslänge
E3F5-  29 0F       AND   #$0F
E3F7-  A8          TAY                  ; Namenslänge
E3F8-  18          CLC
E3F9-  6D BA FE    ADC   $FEBA
E3FC-  AA          TAX
E3FD-  BD 00 D7    LDA   $D700,X
E400-  99 5F FE    STA   $FE5F,Y        ; der Name wird auf $FE60-$FE6E kopiert
E403-  CA          DEX
E404-  88          DEY
E405-  D0 F6       BNE   $E3FD
E407-  A0 03       LDY   #$03
E409-  B1 40       LDA   ($40),Y
E40B-  8D 7D FE    STA   $FE7D          ; $FE7D: ACCESS
E40E-  C8          INY
E40F-  B1 40       LDA   ($40),Y
E411-  8D 6F FE    STA   $FE6F          ; $FE6F: FILE TYPE
E414-  C8          INY
E415-  B1 40       LDA   ($40),Y
E417-  99 79 FE    STA   $FE79,Y        ; $FE7E/7F: AUX TYPE
E41A-  C0 06       CPY   #$06
E41C-  D0 F6       BNE   $E414
E41E-  AD F0 FD    LDA   $FDF0
E421-  8D 7B FE    STA   $FE7B          ; $FE7B/7C: ProDOS VERSION (00 00)
E424-  AD F1 FD    LDA   $FDF1
E427-  8D 7C FE    STA   $FE7C
E42A-  A9 01       LDA   #$01
E42C-  8D 72 FE    STA   $FE72          ; $FE72/73: BLOCKS USED (00 01)
E42F-  AD 5A FE    LDA   $FE5A
E432-  8D 84 FE    STA   $FE84          ; $FE84/85: Nummer des Key Blocks der
E435-  AD 5B FE    LDA   $FE5B            Directory, in dem dieser File steht
E438-  8D 85 FE    STA   $FE85
```

– File Info ist erstellt: Subdirectory Key oder leerer Datenblock?

```
E43B-  AD 5F FE    LDA   $FE5F          ; Storage Type/Namenslänge
E43E-  29 E0       AND   #$E0
```

```
E440-  F0 37     BEQ  $E479      ; Sprung für Type $10 (Datenfile)
E442-  A2 1E     LDX  #$1E
E444-  BD 5F FE  LDA  $FE5F,X
E447-  9D 04 DC  STA  $DC04,X    ; Name..CREATION DATE&TIME werden kopiert
E44A-  CA        DEX
E44B-  10 F7     BPL  $E444
E44D-  49 30     EOR  #$30       ; das letzte Byte ist $FE4F:
E44F-  8D 04 DC  STA  $DC04      ; $D0 EOR $30=>$E0: Storage Typ Key Block
E452-  A2 07     LDX  #$07
E454-  BD E6 FD  LDA  $FDE6,X    ; „uHUSTON!"
E457-  9D 14 DC  STA  $DC14,X      ACCESS: $C3, EOF: $200, Eintragslänge im
E45A-  BD F0 FD  LDA  $FDF0,X      Parent: $27, Einträge pro Block $0D
E45D-  9D 20 DC  STA  $DC20,X    ; VERSION: 00 00
E460-  CA        DEX
E461-  10 F1     BPL  $E454
E463-  A2 02     LDX  #$02
E465-  BD 5C FE  LDA  $FE5C,X    ; Parent Pointer und Eintragsnummer
E468-  9D 27 DC  STA  $DC27,X      im Parent Directory Block
E46B-  CA        DEX
E46C-  10 F7     BPL  $E465
E46E-  AD 51 FE  LDA  $FE51      ; Länge eines Eintrags im Parent
E471-  8D 2A DC  STA  $DC2A
E474-  A9 02     LDA  #$02       ; EOF: $200 ($FE74/76 = 00/00)
E476-  8D 75 FE  STA  $FE75
```

– Sprung hierher, wenn Datenfile: der Block enthält dann nur Nullen

```
E479-  20 B6 EA  JSR  $EAB6      ; ALLOCATE NEXT BLOCK
E47C-  B0 37     BCS  $E4B5
E47E-  8D 70 FE  STA  $FE70      ; Nummer des gefundenen freien Blocks
E481-  8C 71 FE  STY  $FE71      ; Hibyte
E484-  85 46     STA  $46
E486-  84 47     STY  $47
E488-  20 EA EB  JSR  $EBEA      ; der eben erstellte Block wird
E48B-  B0 28     BCS  $E4B5        geschrieben
E48D-  EE 53 FE  INC  $FE53      ; Nummer der aktiven DIR-Einträge => +1
E490-  D0 03     BNE  $E495
E492-  EE 54 FE  INC  $FE54
E495-  20 B6 E4  JSR  $E4B6      ; ENRTY & UPDATE, geht bis zur VolDIR
E498-  B0 1B     BCS  $E4B5
E49A-  4C 93 EB  JMP  $EB93      ; Rewrite der VBM
```

– Berechnung der Speicheradresse eines Directory-Eintrags

Xref: $E4E4 – ENTRY & UPDATE (neuer Eintrag)
$E54F – ENTRY & UPDATE (Loop – Lokalisieren des Eintrags im Parent)
$F595 – FLUSH (Zusammenstellung des File Info vor ENTRY & UPDATE)

```
E49D-  A9 DC     LDA  #$DC       ; Directory-Block auf $DC00/$DD00
E49F-  85 49     STA  $49
E4A1-  A9 04     LDA  #$04       ; Start mit $DC04
E4A3-  AE 5E FE  LDX  $FE5E      ; Eintragsnummer
E4A6-  18        CLC
E4A7-  CA        DEX
E4A8-  F0 09     BEQ  $E4B3
E4AA-  6D 51 FE  ADC  $FE51      ; Länge eines Eintrags
E4AD-  90 F8     BCC  $E4A7
E4AF-  E6 49     INC  $49
E4B1-  B0 F3     BCS  $E4A6
```

```
E4B3-  85 48     STA   $48          ; $0048/49 werden gesetzt
E4B5-  60        RTS
```

**– ENTRY & UPDATE –** Überschreiben eines Eintrags mit gesetzten Parametern (File Info)

Xref: $E495 – CREATE (Datenfile/Subdirectory Key Block)
$F5D8 – FLUSH (Update eines File-Eintrags)
JP $F81E – SET FILE INFO (Überschreiben der File-Parameter, MOD TIME = $00)
JP $F9F0 – DESTROY (setzt Eintrag ungültig: Namenslänge/Storage Type = $00)

```
E4B6-  AD 90 BF  LDA   $BF90
E4B9-  F0 0B     BEQ   $E4C6          ; keine System Time
E4BB-  A2 03     LDX   #$03
E4BD-  BD 90 BF  LDA   $BF90,X
E4C0-  9D 80 FE  STA   $FE80,X        ; System Time wird kopiert
E4C3-  CA        DEX
E4C4-  10 F7     BPL   $E4BD
```

Xref: $F81B – SET FILE INFO (MOD TIME im Parameterblock spezifiziert)
$F939 – RENAME (Überschreiben des File-Eintrags mit neuem Namen)

```
E4C6-  AD 7D FE  LDA   $FE7D          ; ACCESS des Files
E4C9-  0D B4 FE  ORA   $FEB4          ; internes Backup-Bit, außer bei SET F'INFO
E4CC-  8D 7D FE  STA   $FE7D                  immer $20 (s. „Spezielles")
E4CF-  AD 59 FE  LDA   $FE59
E4D2-  8D 30 BF  STA   $BF30          ; Unitnummer
E4D5-  AD 5C FE  LDA   $FE5C
E4D8-  85 46     STA   $46            ; Blocknummer des Directory-Blocks, in dem
E4DA-  AD 5D FE  LDA   $FE5D            der File-Eintrag steht
E4DD-  85 47     STA   $47
E4DF-  20 EE EB  JSR   $EBEE          ; liest den Block auf $DC00
E4E2-  B0 D1     BCS   $E4B5
E4E4-  20 9D E4  JSR   $E49D          ; setzt $0048/49 auf Startadresse des Eintrags
E4E7-  AC 51 FE  LDY   $FE51          ; Länge des Eintrags
E4EA-  88        DEY
E4EB-  B9 5F FE  LDA   $FE5F,Y
E4EE-  91 48     STA   ($48),Y        ; der gesamte Eintrag wird kopiert
E4F0-  88        DEY
E4F1-  10 F8     BPL   $E4EB
E4F3-  AD 5A FE  LDA   $FE5A          ; ist dieser Block der Key Block der DIR?
E4F6-  C5 46     CMP   $46
E4F8-  D0 07     BNE   $E501          ; nein
E4FA-  AD 5B FE  LDA   $FE5B
E4FD-  C5 47     CMP   $47
E4FF-  F0 14     BEQ   $E515          ; ja
E501-  20 EA EB  JSR   $EBEA          ; nicht der Key Block: der Block wird
E504-  B0 AF     BCS   $E4B5            zurückgeschrieben und die Nummer des
E506-  AD 5A FE  LDA   $FE5A            Key Blocks wird gesetzt
E509-  85 46     STA   $46
E50B-  AD 5B FE  LDA   $FE5B
E50E-  85 47     STA   $47
E510-  20 EE EB  JSR   $EBEE          ; liest den Block auf $DC00
E513-  B0 A0     BCS   $E4B5
E515-  A0 01     LDY   #$01
```

```
E517-  B9 53 FE   LDA  $FE53,Y      ; die Anzahl der aktiven Einträge wird
E51A-  99 25 DC   STA  $DC25,Y        kopiert (+1 für CREATE / -1 für DESTROY)
E51D-  88         DEY
E51E-  10 F7      BPL  $E517
E520-  AD 50 FE   LDA  $FE50        ; ACCESS der Directory - nie verändert
E523-  8D 22 DC   STA  $DC22
E526-  20 EA EB   JSR  $EBEA        ; schreibt den Block zurück
E529-  B0 58      BCS  $E583
E52B-  AD 04 DC   LDA  $DC04        ; war dieser Key Block der VolDIR Key?
E52E-  29 F0      AND  #$F0
E530-  49 F0      EOR  #$F0
E532-  F0 6D      BEQ  $E5A1        ; ja, RTS
E534-  AD 27 DC   LDA  $DC27
E537-  85 46      STA  $46          ; ansonsten wird der „Parent Block", also
E539-  AD 28 DC   LDA  $DC28          der Block, in dem diese Directory als
E53C-  85 47      STA  $47            Eintrag steht, gesetzt
E53E-  AD 29 DC   LDA  $DC29        ; die Eintragsnummer innerhalb dieses Blocks
E541-  8D 5E FE   STA  $FE5E
E544-  AD 2A DC   LDA  $DC2A        ; und die Eintragslänge
E547-  8D 51 FE   STA  $FE51
E54A-  20 EE EB   JSR  $EBEE        ; der Block wird gelesen
E54D-  B0 34      BCS  $E583
E54F-  20 9D E4   JSR  $E49D        ; setzt $0048/49 auf Startadresse des Eintrags
E552-  AD 90 BF   LDA  $BF90
E555-  F0 0D      BEQ  $E564        ; keine System Time
E557-  A2 03      LDX  #$03
E559-  A0 24      LDY  #$24
E55B-  BD 90 BF   LDA  $BF90,X      ; die System Time wird in MOD DATE & TIME
E55E-  91 48      STA  ($48),Y        kopiert, das geschieht auch für RENAME,
E560-  88         DEY                 d.h. bei RENAME/SET F'INFO wird zwar die
E561-  CA         DEX                 MOD TIME des Files nicht verändert, aber
E562-  10 F7      BPL  $E55B          die MOD TIME einer Subdirectory schon!
E564-  20 EA EB   JSR  $EBEA        ; schreibt den Block zurück
E567-  B0 1A      BCS  $E583
E569-  A0 25      LDY  #$25
E56B-  B1 48      LDA  ($48),Y
E56D-  C8         INY
E56E-  C5 46      CMP  $46          ; war dieser Block der Key Block der DIR?
E570-  85 46      STA  $46
E572-  D0 06      BNE  $E57A        ; nein
E574-  B1 48      LDA  ($48),Y
E576-  C5 47      CMP  $47
E578-  F0 B1      BEQ  $E52B        ; ja, Loop (Test auf VolDIR Key)
E57A-  B1 48      LDA  ($48),Y
E57C-  85 47      STA  $47          ; sonst wird die Blocknummer des Key Blocks
E57E-  20 EE EB   JSR  $EBEE          gesetzt, der Key Block wird gelesen
E581-  90 A8      BCC  $E52B        ; Loop (Test auf VolDIR Key)
E583-  60         RTS

E584-  A9 52      LDA  #$52         ; ERR=52: NOT A PRODOS VOLUME
E586-  38         SEC                 (Exit von $E588)
E587-  60         RTS
```

– Prüfung, ob der geladene Key Block ein ProDOS (Vol)DIR Key Block ist

Xref: $E8DC – FIND FILE ENTRY (Absuchen aller Units)

```
E588-  AD 00 DC   LDA  $DC00        ; Backward Reference (lo)
E58B-  CD EE FD   CMP  $FDEE        ; = 0
E58E-  D0 F4      BNE  $E584
E590-  AD 01 DC   LDA  $DC01        ; Backward Reference (hi)
E593-  CD EF FD   CMP  $FDEF        ; = 0
E596-  D0 EC      BNE  $E584
```

```
E598-   AD 04 DC    LDA   $DC04
E59B-   29 E0       AND   #$E0          ; muß Storage Type $F oder $E haben
E59D-   C9 E0       CMP   #$E0
E59F-   D0 E3       BNE   $E584
E5A1-   18          CLC
E5A2-   60          RTS
```

– **GET FILE** (der File wird gesucht und die gefundenen Parameter werden kopiert)

Xref: $E144 – SET PREFIX (nicht PREFIX „/")
$EEC6 – OPEN
$F785 – GET FILE INFO
$F7EA – SET FILE INFO
$F8CB – RENAME (GET altes File)
$F966 – DESTROY

```
E5A3-   20 B6 E5    JSR   $E5B6         ; sucht/lädt den DIR-Block, in dem der
E5A6-   B0 0D       BCS   $E5B5           Eintrag steht
```

Xref: $F59B – FLUSH (DIR-Block bereits gefunden, die (alten) Parms werden kopiert)

```
E5A8-   AC 51 FE    LDY   $FE51         ; Länge eines Eintrags
E5AB-   B1 48       LDA   ($48),Y
E5AD-   99 5F FE    STA   $FE5F,Y       ; der Eintrag wird kopiert
E5B0-   88          DEY
E5B1-   10 F8       BPL   $E5AB
E5B3-   A9 00       LDA   #$00          ; überflüssig, nirgends ausgewertet
E5B5-   60          RTS
```

– **FIND FILE ENTRY** (Suche entlang des Pathname und Laden des Directory-Blocks)

Xref: $E302 – CREATE (Verifizierung, daß der neue File nicht schon existiert)
$E5A3 – GET FILE (s.o.)
$F821 – RENAME (Verifizierung, daß der „alte" File existiert)
$F8BB – RENAME (Verifizierung, daß der „neue" File nicht schon existiert)

– Suchen/Laden des entsprechenden DIR-Blocks

```
E5B6-   20 3A E7    JSR   $E73A         ; Suche des Volumes und Lesen
E5B9-   B0 52       BCS   $E60D           der Vol/Prefix DIR (Key Block)
E5BB-   D0 28       BNE   $E5E5         ; Sprung, wenn Pathname nicht zu Ende
```

– Pathname endet in einer Volume Directory
Wenn die VolDIR gelesen werden soll (z.B. OPEN + READ oder GET F'INFO), werden die Parameter, die in einer VolDIR nicht enthalten sind, von einer Tabelle kopiert:
Nummer des ersten Blocks: 00 02 / EOF $800 / Blocks used: 00 04
Danach Exit mit ERR#40 (INVALID PATHNAME).

Bei CREATE/DESTROY führt dieser Error zum Abbruch, SET PREFIX/RENAME benutzen diese Fehlerbedingung zur Unterscheidung von Volume/Subdirectories.

```
E5BD-   AD 35 E6    LDA   $E635        ; $DC
E5C0-   85 49       STA   $49
E5C2-   A9 04       LDA   #$04
E5C4-   85 48       STA   $48          ; $0048/49: $DC04
E5C6-   A0 1F       LDY   #$1F
E5C8-   B1 48       LDA   ($48),Y
E5CA-   99 5F FE    STA   $FE5F,Y      ; zuerst Kopie des gesamten Keys
E5CD-   88          DEY
E5CE-   C0 17       CPY   #$17
E5D0-   D0 F6       BNE   $E5C8
E5D2-   B9 E8 FD    LDA   $FDE8,Y      ; danach „Fälschen" der fehlenden
E5D5-   99 5F FE    STA   $FE5F,Y        Parameter
E5D8-   88          DEY
E5D9-   C0 0F       CPY   #$0F
E5DB-   D0 F5       BNE   $E5D2
E5DD-   A9 D0       LDA   #$D0
E5DF-   8D 5F FE    STA   $FE5F        ; Storage Type => $D (Subdirectory)
E5E2-   A9 40       LDA   #$40         ; ERR#40: INVALID PATHNAME
E5E4-   60          RTS                ; mit Carry gesetzt von „CPY #$0F"
```

– Suche entlang des Pathnames

```
E5E5-   A9 00       LDA   #$00         ; + Suche nach einem freien Dir-Eintrag
E5E7-   8D 9B FE    STA   $FE9B          für CREATE: bis jetzt keiner gefunden
E5EA-   38          SEC                ; gesetzt: Überspringen des Keys in $D8D8
```

– Suche innerhalb einer Directory (mehrere Blocks)

```
E5EB-   A9 00       LDA   #$00
E5ED-   8D 97 FE    STA   $FE97        ; Zähler: Eintrag $00
E5F0-   20 E3 E6    JSR   $E6E3        ; sucht Namen + freien Eintrag
E5F3-   90 65       BCC   $E65A        ; Name gefunden. Nächster Name im Path
E5F5-   AD 98 FE    LDA   $FE98
E5F8-   ED 97 FE    SBC   $FE97        ; nicht gefunden: die Anzahl der gefundenen
E5FB-   90 09       BCC   $E606          Einträge (Zähler $FE97) in diesem Block
E5FD-   D0 10       BNE   $E60F          wird von der Gesamtzahl der aktiven
E5FF-   CD 99 FE    CMP   $FE99          Einträge abgezogen
E602-   F0 2A       BEQ   $E62E        ; ok. Kein Eintrag mehr übrig => NOT FOUND
E604-   D0 09       BNE   $E60F
E606-   CE 99 FE    DEC   $FE99
E609-   10 04       BPL   $E60F        ; noch Einträge übrig: nächster Block
E60B-   A9 51       LDA   #$51         ; ERR#51: FILE STRUCTURE DAMAGED
E60D-   38          SEC                  (mehr Einträge gezählt, als vorhanden
E60E-   60          RTS                  sein sollten)
E60F-   8D 98 FE    STA   $FE98        ; Anzahl der übrigen aktiven Einträge
E612-   AD 35 E6    LDA   $E635
E615-   85 49       STA   $49          ; $0048/49 zurück auf $DC04
E617-   AD 02 DC    LDA   $DC02
E61A-   D0 05       BNE   $E621        ; noch DIR-Blocks übrig
E61C-   CD 03 DC    CMP   $DC03
E61F-   F0 EA       BEQ   $E60B        ; keine Blocks mehr, aber noch Einträge!
E621-   85 46       STA   $46
E623-   AD 03 DC    LDA   $DC03
E626-   85 47       STA   $47
E628-   20 EE EB    JSR   $EBEE        ; Lesen des nächsten Blocks der Directory
E62B-   90 BE       BCC   $E5EB        ; Loop
E62D-   60          RTS
```

– alle aktiven Einträge heruntergezählt. „Patch" für CREATE

```
E62E-  AD 9B FE   LDA   $FE9B      ; freier Eintrag gefunden (<> $00)?
E631-  D0 1B      BNE   $E64E
E633-  AD 02 DC   LDA   $DC02      ; der letzte Block enthält 00 00
E636-  D0 05      BNE   $E63D        als Forward Reference
E638-  CD 03 DC   CMP   $DC03
E63B-  F0 11      BEQ   $E64E      ; war der letzte Block in dieser Dir
E63D-  8D 5C FE   STA   $FE5C      ; ansonsten muß der nächste Dir-Block
E640-  AD 03 DC   LDA   $DC03        leer sein. $FE5C/5D werden auf die
E643-  8D 5D FE   STA   $FE5D        Blocknummer gesetzt.
E646-  A9 01      LDA   #$01
E648-  8D 5E FE   STA   $FE5E      ; Nummer des „gefundenen" freien Eintrags
E64B-  8D 9B FE   STA   $FE9B      ; und Flag „freier Eintrag gefunden"
E64E-  20 7B E7   JSR   $E77B      ; Test: ist das der letzte Name im Path?
E651-  38         SEC
E652-  F0 03      BEQ   $E657      ; Sprung, wenn am Ende des Path
E654-  A9 44      LDA   #$44       ; ERR#44: DIRECTORY NOT FOUND
E656-  60         RTS
E657-  A9 46      LDA   #$46       ; ERR#46: FILE NOT FOUND
E659-  60         RTS
```

– Name im Directory gefunden

```
E65A-  20 74 E7   JSR   $E774      ; Index auf nächsten Namen im Path
E65D-  F0 6C      BEQ   $E6CB      ; Path-Ende: Speichern der Blockno + Exit
E65F-  A0 00      LDY   #$00
E661-  B1 48      LDA   ($48),Y    ; der Pathname ist nicht zu Ende: der
E663-  29 F0      AND   #$F0         gefundene Eintrag muß der Eintrag für
E665-  C9 D0      CMP   #$D0         eine Subdirectory ($D) sein
E667-  D0 E8      BNE   $E651      ; => ERR#46: DIRECTORY NOT FOUND
E669-  A0 11      LDY   #$11       ; ist Subdirectory: die Nummer des
E66B-  B1 48      LDA   ($48),Y      ersten („Key") Blocks wird auf $0046/47
E66D-  85 46      STA   $46          und auf $FE5A/5B („Key Blocknummer")
E66F-  C8         INY                kopiert.
E670-  8D 5A FE   STA   $FE5A
E673-  B1 48      LDA   ($48),Y
E675-  85 47      STA   $47
E677-  8D 5B FE   STA   $FE5B
E67A-  20 EE EB   JSR   $EBEE      ; der Key Block wird gelesen
E67D-  B0 24      BCS   $E6A3
E67F-  AD 25 DC   LDA   $DC25      ; Anzahl der aktiven Einträge =>
E682-  8D 98 FE   STA   $FE98      ; Herunterzähler für den Such-Loop
E685-  AD 26 DC   LDA   $DC26
E688-  8D 99 FE   STA   $FE99
E68B-  AD 21 DC   LDA   $DC21      ; MLI VERSION: muß $00 sein, sonst
E68E-  D0 11      BNE   $E6A1        ERR#4A: INCOMPATIBLE FILE FORMAT
E690-  AD 14 DC   LDA   $DC14    ; $75 im Subdir Key (s. „Spezielles")
E693-  A2 00      LDX   #$00
E695-  38         SEC
E696-  2A         ROL
E697-  90 01      BCC   $E69A      ; Zählung der gesetzten Bits
E699-  E8         INX
E69A-  0A         ASL
E69B-  D0 FA      BNE   $E697
E69D-  E0 05      CPX   #$05       ; $75 => 5 Bits gesetzt
E69F-  F0 04      BEQ   $E6A5
E6A1-  A9 4A      LDA   #$4A       ; ERR#4A: INCOMPATIBLE FILE FORMAT
E6A3-  38         SEC
E6A4-  60         RTS
E6A5-  20 AB E6   JSR   $E6AB      ; Kopiert die Parameter dieser DIR
E6A8-  4C E5 E5   JMP   $E5E5      ; Loop: nächster Name im Path
```

Bei CREATE/DESTROY führt dieser Error zum Abbruch, SET PREFIX/RENAME benutzen diese Fehlerbedingung zur Unterscheidung von Volume/Subdirectories.

```
E5BD-   AD 35 E6    LDA   $E635         ; $DC
E5C0-   85 49       STA   $49
E5C2-   A9 04       LDA   #$04
E5C4-   85 48       STA   $48           ; $0048/49: $DC04
E5C6-   A0 1F       LDY   #$1F
E5C8-   B1 48       LDA   ($48),Y
E5CA-   99 5F FE    STA   $FE5F,Y       ; zuerst Kopie des gesamten Keys
E5CD-   88          DEY
E5CE-   C0 17       CPY   #$17
E5D0-   D0 F6       BNE   $E5C8
E5D2-   B9 E8 FD    LDA   $FDE8,Y       ; danach „Fälschen" der fehlenden
E5D5-   99 5F FE    STA   $FE5F,Y         Parameter
E5D8-   88          DEY
E5D9-   C0 0F       CPY   #$0F
E5DB-   D0 F5       BNE   $E5D2
E5DD-   A9 D0       LDA   #$D0
E5DF-   8D 5F FE    STA   $FE5F         ; Storage Type => $D (Subdirectory)
E5E2-   A9 40       LDA   #$40          ; ERR#40: INVALID PATHNAME
E5E4-   60          RTS                 ; mit Carry gesetzt von „CPY #$0F"
```

– Suche entlang des Pathnames

```
E5E5-   A9 00       LDA   #$00          ; + Suche nach einem freien Dir-Eintrag
E5E7-   8D 9B FE    STA   $FE9B           für CREATE: bis jetzt keiner gefunden
E5EA-   38          SEC                 ; gesetzt: Überspringen des Keys in $D8D8
```

– Suche innerhalb einer Directory (mehrere Blocks)

```
E5EB-   A9 00       LDA   #$00
E5ED-   8D 97 FE    STA   $FE97         ; Zähler: Eintrag $00
E5F0-   20 E3 E6    JSR   $E6E3         ; sucht Namen + freien Eintrag
E5F3-   90 65       BCC   $E65A         ; Name gefunden. Nächster Name im Path
E5F5-   AD 98 FE    LDA   $FE98
E5F8-   ED 97 FE    SBC   $FE97         ; nicht gefunden: die Anzahl der gefundenen
E5FB-   90 09       BCC   $E606           Einträge (Zähler $FE97) in diesem Block
E5FD-   D0 10       BNE   $E60F           wird von der Gesamtzahl der aktiven
E5FF-   CD 99 FE    CMP   $FE99           Einträge abgezogen
E602-   F0 2A       BEQ   $E62E         ; ok. Kein Eintrag mehr übrig => NOT FOUND
E604-   D0 09       BNE   $E60F
E606-   CE 99 FE    DEC   $FE99
E609-   10 04       BPL   $E60F         ; noch Einträge übrig: nächster Block
E60B-   A9 51       LDA   #$51          ; ERR#51: FILE STRUCTURE DAMAGED
E60D-   38          SEC                   (mehr Einträge gezählt, als vorhanden
E60E-   60          RTS                   sein sollten)
E60F-   8D 98 FE    STA   $FE98         ; Anzahl der übrigen aktiven Einträge
E612-   AD 35 E6    LDA   $E635
E615-   85 49       STA   $49           ; $0048/49 zurück auf $DC04
E617-   AD 02 DC    LDA   $DC02
E61A-   D0 05       BNE   $E621         ; noch DIR-Blocks übrig
E61C-   CD 03 DC    CMP   $DC03
E61F-   F0 EA       BEQ   $E60B         ; keine Blocks mehr, aber noch Einträge!
E621-   85 46       STA   $46
E623-   AD 03 DC    LDA   $DC03
E626-   85 47       STA   $47
E628-   20 EE EB    JSR   $EBEE         ; Lesen des nächsten Blocks der Directory
E62B-   90 BE       BCC   $E5EB         ; Loop
E62D-   60          RTS
```

– alle aktiven Einträge heruntergezählt. „Patch" für CREATE

```
E62E-   AD 9B FE    LDA   $FE9B        ; freier Eintrag gefunden (<> $00)?
E631-   D0 1B       BNE   $E64E
E633-   AD 02 DC    LDA   $DC02        ; der letzte Block enthält 00 00
E636-   D0 05       BNE   $E63D          als Forward Reference
E638-   CD 03 DC    CMP   $DC03
E63B-   F0 11       BEQ   $E64E        ; war der letzte Block in dieser Dir
E63D-   8D 5C FE    STA   $FE5C        ; ansonsten muß der nächste Dir-Block
E640-   AD 03 DC    LDA   $DC03          leer sein. $FE5C/5D werden auf die
E643-   8D 5D FE    STA   $FE5D          Blocknummer gesetzt.
E646-   A9 01       LDA   #$01
E648-   8D 5E FE    STA   $FE5E        ; Nummer des „gefundenen" freien Eintrags
E64B-   8D 9B FE    STA   $FE9B        ; und Flag „freier Eintrag gefunden"
E64E-   20 7B E7    JSR   $E77B        ; Test: ist das der letzte Name im Path?
E651-   38          SEC
E652-   F0 03       BEQ   $E657        ; Sprung, wenn am Ende des Path
E654-   A9 44       LDA   #$44         ; ERR#44: DIRECTORY NOT FOUND
E656-   60          RTS
E657-   A9 46       LDA   #$46         ; ERR#46: FILE NOT FOUND
E659-   60          RTS
```

– Name im Directory gefunden

```
E65A-   20 74 E7    JSR   $E774        ; Index auf nächsten Namen im Path
E65D-   F0 6C       BEQ   $E6CB        ; Path-Ende: Speichern der Blockno + Exit
E65F-   A0 00       LDY   #$00
E661-   B1 48       LDA   ($48),Y      ; der Pathname ist nicht zu Ende: der
E663-   29 F0       AND   #$F0           gefundene Eintrag muß der Eintrag für
E665-   C9 D0       CMP   #$D0           eine Subdirectory ($D) sein
E667-   D0 E8       BNE   $E651        ; => ERR#46: DIRECTORY NOT FOUND
E669-   A0 11       LDY   #$11         ; ist Subdirectory: die Nummer des
E66B-   B1 48       LDA   ($48),Y        ersten („Key") Blocks wird auf $0046/47
E66D-   85 46       STA   $46            und auf $FE5A/5B („Key Blocknummer")
E66F-   C8          INY                  kopiert.
E670-   8D 5A FE    STA   $FE5A
E673-   B1 48       LDA   ($48),Y
E675-   85 47       STA   $47
E677-   8D 5B FE    STA   $FE5B
E67A-   20 EE EB    JSR   $EBEE        ; der Key Block wird gelesen
E67D-   B0 24       BCS   $E6A3
E67F-   AD 25 DC    LDA   $DC25        ; Anzahl der aktiven Einträge =>
E682-   8D 98 FE    STA   $FE98        ; Herunterzähler für den Such-Loop
E685-   AD 26 DC    LDA   $DC26
E688-   8D 99 FE    STA   $FE99
E68B-   AD 21 DC    LDA   $DC21        ; MLI VERSION: muß $00 sein, sonst
E68E-   D0 11       BNE   $E6A1          ERR#4A: INCOMPATIBLE FILE FORMAT
E690-   AD 14 DC    LDA   $DC14        ; $75 im Subdir Key (s. „Spezielles")
E693-   A2 00       LDX   #$00
E695-   38          SEC
E696-   2A          ROL
E697-   90 01       BCC   $E69A        ; Zählung der gesetzten Bits
E699-   E8          INX
E69A-   0A          ASL
E69B-   D0 FA       BNE   $E697
E69D-   E0 05       CPX   #$05         ; $75 => 5 Bits gesetzt
E69F-   F0 04       BEQ   $E6A5
E6A1-   A9 4A       LDA   #$4A         ; ERR#4A: INCOMPATIBLE FILE FORMAT
E6A3-   38          SEC
E6A4-   60          RTS
E6A5-   20 AB E6    JSR   $E6AB        ; Kopiert die Parameter dieser DIR
E6A8-   4C E5 E5    JMP   $E5E5        ; Loop: nächster Name im Path
```

– Kopie der Parameter aus einem geladenen Key Block

Xref: $E6A5 – diese Directory ist Parent => Suche nach dem nächsten Namen im Path
$E74F – SEARCH VOLUME
$F57E – FLUSH: (Sub)dir gefunden, Kopie der Parms vor Suche des Filenamens

```
E6AB-   A2 0A       LDX   #$0A          ; DATE & TIME: $FE4A..$FE4D (nie benutzt)
E6AD-   BD 1C DC    LDA   $DC1C,X         VERS:$FE4E/4F ACCESS:$FE50(nie geändert)
E6B0-   9D 4A FE    STA   $FE4A,X         Länge eines Eintrags: $FE51
E6B3-   CA          DEX                   Anzahl Einträge per Block: $FE52
E6B4-   10 F7       BPL   $E6AD          Gesamtzahl aktive Einträge: $FE53/54
E6B6-   AD 04 DC    LDA   $DC04        ; Storage Type
E6B9-   29 F0       AND   #$F0
E6BB-   49 F0       EOR   #$F0
E6BD-   F0 0B       BEQ   $E6CA        ; für VolDIR: RTS
E6BF-   A2 03       LDX   #$03         ; für Subdir:
E6C1-   BD 27 DC    LDA   $DC27,X        Parent Pointer (Blockno Parent): $FE46/47
E6C4-   9D 46 FE    STA   $FE46,X        Eintragsnummer im  Parent-Block: $FE48
E6C7-   CA          DEX                  Länge eines Eintrags im Parent:  $FE49
E6C8-   10 F7       BPL   $E6C1
E6CA-   60          RTS
```

– Speichern der momentanen Eintragsnummer ($01-$0D) und Nummer des Dir-Blocks

Xref: $E6FF – Name im Dir-Block gefunden. ($E6FF vom FIND FILE-Loop aufgerufen).

```
E6CB-   AD 52 FE    LDA   $FE52        ; Anzahl der Einträge per Block
E6CE-   38          SEC
E6CF-   ED 9A FE    SBC   $FE9A        ; Herunterzähler vom Such-Loop
E6D2-   69 00       ADC   #$00         ; + 1 (Carry set) => Bereich $01-$0D
E6D4-   8D 5E FE    STA   $FE5E        ; Nummer des momentanen Eintrags
E6D7-   A5 46       LDA   $46
E6D9-   8D 5C FE    STA   $FE5C        ; Nummer des Directory-Blocks, in dem
E6DC-   A5 47       LDA   $47            der Eintrag gefunden wurde
E6DE-   8D 5D FE    STA   $FE5D
E6E1-   18          CLC
E6E2-   60          RTS
```

– Suche nach einem Namen des Path innerhalb eines Dir-Blocks

Xref: $E5F0 – Absuchen einer Directory in FIND FILE

```
E6E3-   AD 52 FE    LDA   $FE52        ; Anzahl der Einträge per Dir-Block
E6E6-   8D 9A FE    STA   $FE9A        ; Herunterzähler
E6E9-   AD 35 E6    LDA   $E635
E6EC-   85 49       STA   $49
E6EE-   A9 04       LDA   #$04
E6F0-   85 48       STA   $48          ; $0048/49 = $DC04
E6F2-   B0 33       BCS   $E727        ; Aufruf mit SEC: Erster Eintrag (Header)
E6F4-   A0 00       LDY   #$00                   wird übersprungen
E6F6-   B1 48       LDA   ($48),Y      ; Typ/Längenbyte eines Eintrags
E6F8-   D0 0D       BNE   $E707        ; Eintrag ist besetzt: Sprung
E6FA-   AD 9B FE    LDA   $FE9B        ; „Freier Eintrag gefunden"-Flag
E6FD-   D0 28       BNE   $E727        ; Sprung, wenn bereits vorher gesetzt
E6FF-   20 CB E6    JSR   $E6CB        ; Ansonsten Speichern von Blockno/Eintragsno
E702-   EE 9B FE    INC   $FE9B          und Setzen von „Freier Eintrag gefunden"
E705-   D0 20       BNE   $E727        ; „always": NEXT (nächster Eintrag)
```

```
E707-  29 0F      AND   #$0F          ; Sprung hierher, wenn Eintrag besetzt
E709-  EE 97 FE   INC   $FE97         ; Zähler der „aktiven" Einträge
E70C-  8D B8 FE   STA   $FEB8         ; Länge des Namens: Zähler für Vergleich
E70F-  AE BA FE   LDX   $FEBA         ; Index im Pathname
E712-  DD 00 D7   CMP   $D700,X       ; dieselbe Länge?
E715-  D0 10      BNE   $E727         ; nein, Namen müssen verschieden sein: NE
E717-  E8         INX
E718-  C8         INY

E719-  B1 48      LDA   ($48),Y       ; Vergleich mit dem Namen im Path
E71B-  DD 00 D7   CMP   $D700,X
E71E-  D0 07      BNE   $E727
E720-  CE B8 FE   DEC   $FEB8
E723-  D0 F2      BNE   $E717         ; bis herunter zum ersten Zeichen
E725-  18         CLC
E726-  60         RTS                 ; Name bzw. Eintrag gefunden

E727-  CE 9A FE   DEC   $FE9A         ; Loop-Zähler
E72A-  38         SEC
E72B-  F0 F9      BEQ   $E726         ; Name nicht gefunden
E72D-  AD 51 FE   LDA   $FE51         ; Länge eines Eintrags
E730-  18         CLC
E731-  65 48      ADC   $48           ; $0048/49 auf nächsten Eintrag
E733-  90 BB      BCC   $E6F0
E735-  E6 49      INC   $49
E737-  18         CLC
E738-  90 B6      BCC   $E6F0         ; Loop: nächster Name
```

**– GET VOLDIR/PREFIX KEY BLOCK** – Setzen der DIR-Parameter

Xref: $E5B6 – FIND FILE ENTRY

```
E73A-  20 93 E7   JSR   $E793         ; GET VOLUME sucht & lädt VolDIR Key Block
E73D-  B0 53      BCS   $E792
E73F-  A9 00      LDA   #$00          ; Volume gefunden:
E741-  A0 42      LDY   #$42
E743-  99 46 FE   STA   $FE46,Y       ; $FE46..$FE88 werden gelöscht
E746-  88         DEY
E747-  10 FA      BPL   $E743
E749-  AD 30 BF   LDA   $BF30         ; Unitnummer, gesetzt von GET VOLUME
E74C-  8D 59 FE   STA   $FE59
E74F-  20 AB E6   JSR   $E6AB         ; Kopiert DATE & TIME/ACCESS.. aus dem Key
E752-  A0 01      LDY   #$01            Block, bei Subdir auch Parent Pointer
E754-  AE 91 FE   LDX   $FE91         ; VCB-Index, gesetzt von GET VOLUME
E757-  E8         INX
E758-  BD 12 D9   LDA   $D912,X       ; TOTAL BLOCKS auf dem Volume ($D912) und
E75B-  99 57 FE   STA   $FE57,Y         Blocknummer des VolDIR Key Blocks ($D91A)
E75E-  BD 1A D9   LDA   $D91A,X         müssen aus dem VCB kopiert werden, eine
E761-  99 55 FE   STA   $FE55,Y         PREFIXed Subdirectory enthält diese
E764-  B9 46 00   LDA   $0046,Y         Informationen nicht!
E767-  99 5A FE   STA   $FE5A,Y       ; Blocknummer dieses Keys (f. SubDIRs)
E76A-  B9 53 FE   LDA   $FE53,Y       ; Anzahl der aktiven Einträge als
E76D-  99 98 FE   STA   $FE98,Y       ; Herunterzähler für die Suche nach dem
E770-  CA         DEX                   nächsten Namen (= Eintrag) im Path
E771-  88         DEY
E772-  10 E4      BPL   $E758
```

– Überspringen des momentanen Namen im Path

Xref: $E65A – Name in Directory gefunden

```
E774-  20 7B E7   JSR  $E77B        ; berechnet den Index des nächsten Namens
E777-  8C BA FE   STY  $FEBA        ; Index wird gesetzt
E77A-  60         RTS               ; Z-Flag gesetzt bei Path-Ende
```

– Berechnung des Index des nächsten Namens im Path

Xref: $E64E – Test auf Ende des Pathname (=> DIR oder FILE NOT FOUND)
$E774 – Überspringen („Abhaken") des momentanen Namens im Path

```
E77B-  AC BA FE   LDY  $FEBA        ; Startindex des momentanen Namens
E77E-  B9 00 D7   LDA  $D700,Y      ; = Längenbyte
E781-  38         SEC               ; Name + Längenbyte = Namenslänge + 1
E782-  6D BA FE   ADC  $FEBA        ; => 00 bei Namensende
E785-  A8         TAY
E786-  D0 06      BNE  $E78E        ; nicht zu Ende
E788-  AD 30 BF   LDA  $BF30
E78B-  8D 9F FE   STA  $FE9F        ; Namensende: Unitnummer wird gespeichert
E78E-  B9 00 D7   LDA  $D700,Y      ; Länge des nächsten Namens
E791-  18         CLC
E792-  60         RTS
```

**– SEARCH VOLUME/PREFIXED DIRECTORY**

Xref: $E73A – GET VOLUME/PREFIXED DIRECTORY KEY BLOCK

```
E793-  A9 00      LDA  #$00
E795-  AC 9A BF   LDY  $BF9A        ; System PREFIX
E798-  2C BC FE   BIT  $FEBC        ; FULL NAME Flag
E79B-  10 01      BPL  $E79E        ; Name nicht vollständig
E79D-  A8         TAY
E79E-  8C BB FE   STY  $FEBB        ; $00 (voller Name) oder Start SYS PREFIX
E7A1-  8D 30 BF   STA  $BF30        ; = $00
```

– Absuchen des VCB nach dem Volume-Namen

```
E7A4-  48         PHA               ; Loop im VCB: Start mit Index 00
E7A5-  AA         TAX
E7A6-  BD 00 D9   LDA  $D900,X      ; Namenslänge eines VCB-Eintrags
E7A9-  D0 0C      BNE  $E7B7        ; Eintrag enthält gültige Daten
E7AB-  AC BB FE   LDY  $FEBB        ; VCB-Eintrag unbesetzt: NE
E7AE-  68         PLA
E7AF-  18         CLC
E7B0-  69 20      ADC  #$20         ; nächster Index
E7B2-  90 F0      BCC  $E7A4        ; nächster Eintrag
E7B4-  4C 08 E8   JMP  $E808        ; nicht gefunden

E7B7-  8D B8 FE   STA  $FEB8        ; Namenslänge: Herunterzähler
E7BA-  D9 00 D7   CMP  $D700,Y      ; Name im Path (Start mit Namenslänge)
E7BD-  D0 EC      BNE  $E7AB        ; ungleich: NE
E7BF-  E8         INX
E7C0-  C8         INY
E7C1-  BD 00 D9   LDA  $D900,X
E7C4-  CE B8 FE   DEC  $FEB8
E7C7-  10 F1      BPL  $E7BA
E7C9-  68         PLA               ; Namen sind gleich (incl. Länge)
E7CA-  AA         TAX               ; = Startindex im VCB für diesen Eintrag
E7CB-  8D 91 FE   STA  $FE91        ; VCB-Index wird gesetzt
```

```
E7CE-  BD 10 D9   LDA  $D910,X
E7D1-  8D 30 BF   STA  $BF30        ; Unitnummer für diesen Eintrag
E7D4-  A9 00      LDA  #$00
E7D6-  85 47      STA  $47
E7D8-  A9 02      LDA  #$02         ; Blocknummer 00 02 = VolDIR Key Block
E7DA-  85 46      STA  $46
E7DC-  AD BB FE   LDA  $FEBB        ; FULL NAME Flag
E7DF-  A8         TAY
E7E0-  8D BA FE   STA  $FEBA
E7E3-  F0 10      BEQ  $E7F5        ; $00: Full Name, VolDIR Key wird gelesen
E7E5-  38         SEC               ; <> $00 => SYS PREFIX gesezt, $FEA1/A2 =>
E7E6-  79 00 D7   ADC  $D700,Y        Blockno PREFIX DIR. Der Index im Pathname
E7E9-  90 F4      BCC  $E7DF          wird solange erhöht, bis der letzte
E7EB-  AD A0 FE   LDA  $FEA0          Name im PREFIX erreicht ist. Die PREFIX
E7EE-  85 46      STA  $46            DIR muß ebenfalls diesen Namen haben
E7F0-  AD A1 FE   LDA  $FEA1
E7F3-  85 47      STA  $47          ; Setzen der PREFIX Blocknummer

E7F5-  20 EE EB   JSR  $EBEE        ; der Key der (PREFIX/VOL)DIR wird gelesen
E7F8-  B0 06      BCS  $E800
E7FA-  20 9E E8   JSR  $E89E        ; Test auf ProDOS-Directory/Namensvergleich
E7FD-  B0 01      BCS  $E800        ; Stimmt nicht überein (Disk gewechselt)
E7FF-  60         RTS               ; ok. (PREFIX) DIR gefunden & Key geladen
```

– Name stimmt nicht überein (Vol/PREFIX DIR)

Xref: $E7FD – Disk wurde gewechselt oder Lesefehler

```
E800-  AE 91 FE   LDX  $FE91        ; VCB-Index
E803-  BD 11 D9   LDA  $D911,X      ; OPEN File auf diesem Volume?
E806-  30 19      BMI  $E821        ; Ja, es darf nichts im VCB geändert werden!
```

Xref: $E7B4 – Volume nicht im VCB gefunden

```
E808-  AD BB FE   LDA  $FEBB        ; Reload des Pathname-Starts
E80B-  8D BA FE   STA  $FEBA
E80E-  20 64 E8   JSR  $E864        ; kopiert die DEVICE TABLE auf $FECA...
E811-  AD 30 BF   LDA  $BF30        ; wurde bereits versucht, von einem Volume
E814-  D0 0F      BNE  $E825          zu lesen (Disk gewechselt)?. Wenn ja,
                                      dann Start mit diesem Unit
E816-  AE 31 BF   LDX  $BF31        ; ansonsten Start von „oben herunter"
E819-  BD CA FE   LDA  $FECA,X      ; von $E864 kopierte DEVICE TABLE
E81C-  D0 0A      BNE  $E828
E81E-  CA         DEX               ; Unit bereits erledigt (s. $E835)
E81F-  10 F8      BPL  $E819        ; Loop bis zum letzten („untersten") Unit
E821-  A9 45      LDA  #$45         ; ERR#45: VOLUME NOT FOUND
E823-  38         SEC
E824-  60         RTS

E825-  AE 31 BF   LDX  $BF31        ; Einsprung von $E814

E828-  DD CA FE   CMP  $FECA,X
E82B-  F0 05      BEQ  $E832        ; wenn via $E814: Suche nach dem zuletzt
E82D-  CA         DEX                 benutzten Unit
E82E-  10 F8      BPL  $E828
E830-  30 EF      BMI  $E821

E832-  8D 30 BF   STA  $BF30        ; die Unitnummer wird gesetzt
E835-  A9 00      LDA  #$00
E837-  9D CA FE   STA  $FECA,X      ; und das DEVICE aus der Tabelle gelöscht
E83A-  20 76 E8   JSR  $E876        ; Suche nach einem freien Eintrag oder
E83D-  B0 24      BCS  $E863          einem Eintrag mit dieser Unitno
```

```
E83F-   AE 91 FE    LDX   $FE91          ; von $E876 gesetzter Index
E842-   BD 00 D9    LDA   $D900,X
E845-   F0 05       BEQ   $E84C          ; Sprung, wenn der Eintrag frei ist
E847-   BD 11 D9    LDA   $D911,X        ; Eintrag hat dieselbe Unitno. OPEN File?
E84A-   30 CA       BMI   $E816          ; Wenn OPEN File, nächstes Unit

E84C-   A9 00       LDA   #$00           ; freier VCB-Eintrag oder kein OPEN File:
E84E-   85 47       STA   $47              Block 00 02 (VolDIR Key) wird gesetzt
E850-   A9 02       LDA   #$02
E852-   85 46       STA   $46
E854-   20 EE EB    JSR   $EBEE          ; und gelesen
E857-   B0 BD       BCS   $E816
E859-   20 C4 E8    JSR   $E8C4          ; erstellt den VCB-Eintrag mit den Daten
E85C-   B0 B8       BCS   $E816            des gelesenen Key Blocks/Test ProDOS DIR
E85E-   20 9E E8    JSR   $E89E          ; Vergleich Key/Name im Path
E861-   B0 B3       BCS   $E816          ; nicht gleich: Loop
E863-   60          RTS                  ; ok. Gesuchtes Volume gefunden
```

– Erstellt eine Kopie aller Units (DEVICE TABLE) von der System Global Page

Xref: $E26C – ONLINE ALL
$E80E – GET VOLUME – Absuchen aller Units nach einem bestimmten Volume

```
E864-   AE 31 BF    LDX   $BF31          ; DEVNUM – Anzahl der Units
E867-   BD 32 BF    LDA   $BF32,X        ; DEVICE TABLE auf der SGP
E86A-   29 F0       AND   #$F0           ; Ausblenden der „Characteristics",
E86C-   9D CA FE    STA   $FECA,X          übrig bleibt DSSS (Drive/Slot)
E86F-   CA          DEX
E870-   10 F5       BPL   $E867
E872-   AE 31 BF    LDX   $BF31
E875-   60          RTS
```

–Sucht nach einem (freien) VCB-Eintrag und setzt den VCB-Index
Aufruf mit gesetzter Unitnummer. Wird kein Eintrag für diese Unitno gefunden, dann Suche nach einem freien Eintrag. Möglicher Fehler: VCB TABLE FULL.

Xref: $E28E – ONLINE SPECIFIC
$E83A – GET VOLUME – Volume in VCB nicht gefunden, internes ONLINE ALL
$F4DA – CLOSE single file: Suche nach dem VCB-Eintrag, um die Anzahl der OPEN Files um eins zu erniedrigen.

```
E876-   A9 00       LDA   #$00
E878-   A0 FF       LDY   #$FF           ; Annahme: keine freien Einträge
E87A-   AA          TAX
E87B-   BD 10 D9    LDA   $D910,X        ; zugeordnete Unitnummer
E87E-   CD 30 BF    CMP   $BF30          ; ist die gesuchte Unitno?
E881-   D0 05       BNE   $E888
E883-   8E 91 FE    STX   $FE91          ; ja, VCB-Index wird gesetzt
E886-   18          CLC
E887-   60          RTS
E888-   BD 00 D9    LDA   $D900,X        ; Ist der Eintrag frei?
E88B-   D0 04       BNE   $E891          ; nein: NEXT
E88D-   C8          INY                  ; ja: freier Eintrag gefunden (Y <> $FF)
E88E-   8E 91 FE    STX   $FE91          ; FCB-Index wird gesetzt
E891-   8A          TXA                  ; nächster Eintrag (NEXT)
E892-   18          CLC
```

```
E893-  69 20      ADC   #$20
E895-  D0 E3      BNE   $E87A
E897-  98         TYA
E898-  30 01      BMI   $E89B       ; kein freier Eintrag gefunden (Y = $FF)
E89A-  18         CLC
E89B-  A9 55      LDA   #$55        ; ERR#55: VCB TABLE FULL
E89D-  60         RTS
```

– Prüft den geladenen Block auf ProDOS (Sub)DIR Key & Namensvergleich

Xref: $E7FA – FIND FILE – Volume oder PREFIX Dir gefunden
$E85E – GET VOLUME – Volume nicht im VCB gefunden, internes ONLINE ALL

```
E89E-  A2 00      LDX   #$00
E8A0-  AC BA FE   LDY   $FEBA       ; Startindex des momentanen Namens im Path
E8A3-  AD 04 DC   LDA   $DC04       ; Storage Type
E8A6-  C9 E0      CMP   #$E0        ; muß $E oder $F sein
E8A8-  90 07      BCC   $E8B1
E8AA-  29 0F      AND   #$0F        ; => Namenslänge
E8AC-  8D B8 FE   STA   $FEB8       ; Herunterzähler
E8AF-  D0 05      BNE   $E8B6       ; muß > 0 sein
E8B1-  38         SEC
E8B2-  60         RTS
E8B3-  BD 04 DC   LDA   $DC04,X     ; Vergleich Key - Name im Path
E8B6-  D9 00 D7   CMP   $D700,Y
E8B9-  D0 F6      BNE   $E8B1
E8BB-  E8         INX
E8BC-  C8         INY
E8BD-  CE B8 FE   DEC   $FEB8
E8C0-  10 F1      BPL   $E8B3
E8C2-  18         CLC               ; ok. Namen sind gleich
E8C3-  60         RTS
```

– Kopiert die Parameter der gelesenen VolDIR in den VCB

Xref: $E859 – GET VOLUME – Volume nicht in VCB gefunden, internes ONLINE ALL

```
E8C4-  AE 91 FE   LDX   $FE91       ; VCB-Index von SCAN FCB ($E876)
E8C7-  BD 00 D9   LDA   $D900,X
E8CA-  F0 05      BEQ   $E8D1       ; war ein freier Eintrag
E8CC-  20 29 E9   JSR   $E929       ; ansonsten Vergleich VCB - Key
E8CF-  90 57      BCC   $E928       ; sind gleich - kein Update
```

Xref: $E2AD – ONLINE: der alte VCB-Eintrag ist leer oder hat keinen OPEN File

```
E8D1-  A9 00      LDA   #$00
E8D3-  A0 1F      LDY   #$1F        ; der gesamte VCB-Eintrag wird gelöscht
E8D5-  9D 00 D9   STA   $D900,X
E8D8-  E8         INX
E8D9-  88         DEY
E8DA-  10 F9      BPL   $E8D5

E8DC-  20 88 E5   JSR   $E588       ; Prüfung auf ProDOS VolDIR Key Block
E8DF-  B0 47      BCS   $E928       ; kein ProDOS-Volume
E8E1-  20 4A E9   JSR   $E94A       ; Absuchen des VCB - existiert der NAME(!)?
E8E4-  B0 41      BCS   $E927         CLC/RTS wenn der Volume-Name mit einem
```

```
E8E6-   AD 04 DC    LDA   $DC04          OPEN File existiert. $FEB5/B6 sind dann
E8E9-   29 0F       AND   #$0F           auf Flag/VCB-Index gesetzt
E8EB-   A8          TAY                ; Namenslänge
E8EC-   48          PHA
E8ED-   0D 91 FE    ORA   $FE91         ; VCB-Index + Namenslänge
E8F0-   AA          TAX
E8F1-   B9 04 DC    LDA   $DC04,Y
E8F4-   9D 00 D9    STA   $D900,X       ; Kopie des Vol-Namens in den VCB-Eintrag
E8F7-   CA          DEX
E8F8-   88          DEY
E8F9-   D0 F6       BNE   $E8F1
E8FB-   68          PLA                ; Namenslänge
E8FC-   9D 00 D9    STA   $D900,X       ; OHNE Storage Type
E8FF-   AD 30 BF    LDA   $BF30
E902-   9D 10 D9    STA   $D910,X       ; Byte $10: Unitnummer
E905-   AD 29 DC    LDA   $DC29
E908-   9D 12 D9    STA   $D912,X       ; Bytes $12/13: TOTAL BLOCKS
E90B-   AD 2A DC    LDA   $DC2A
E90E-   9D 13 D9    STA   $D913,X
E911-   A5 46       LDA   $46
E913-   9D 16 D9    STA   $D916,X       ; Bytes $16/17: Blocknummer des VolDIR Keys
E916-   A5 47       LDA   $47
E918-   9D 17 D9    STA   $D917,X
E91B-   AD 27 DC    LDA   $DC27
E91E-   9D 1A D9    STA   $D91A,X       ; Bytes $1A/1B: Blocknummer VBM
E921-   AD 28 DC    LDA   $DC28                (erster VBM-Block)
E924-   9D 1B D9    STA   $D91B,X
E927-   18          CLC
E928-   60          RTS
```

– Vergleicht die neu geladene VolDIR mit dem alten VCB-Eintrag

Xref: $E2BC – ONLINE ALL (nur sinnvoll für Volumes mit OPEN File)
$E8CC – Kopieren der gelesenen VolDIR in den VCB
$E94D – Absuchen des gesamten VCB auf Namensbasis

```
E929-   AD 04 DC    LDA   $DC04         ; Storage Type/Namenslänge
E92C-   29 0F       AND   #$0F          ; Namenslänge
E92E-   DD 00 D9    CMP   $D900,X       ; vs. Namenslänge im VCB-Eintrag
E931-   D0 0E       BNE   $E941         ; SEC/RTS
E933-   A8          TAY
E934-   8E 90 FE    STX   $FE90         ; Scratch
E937-   0D 90 FE    ORA   $FE90
E93A-   AA          TAX
E93B-   B9 04 DC    LDA   $DC04,Y       ; Vergleich des gesamten Namens von hinten
E93E-   DD 00 D9    CMP   $D900,X
E941-   38          SEC                ; RTS mit SEC, wenn Namen ungleich
E942-   D0 05       BNE   $E949
E944-   CA          DEX
E945-   88          DEY
E946-   D0 F3       BNE   $E93B
E948-   18          CLC                ; ok. Namen sind gleich
E949-   60          RTS
```

– Absuchen des gesamten VCB auf Namensbasis

Xref: $E8E1 – Update des VCB mit geladener VolDIR

```
E94A-   A9 00       LDA   #$00          ; Start mit erstem VCB-Eintrag (Index)
E94C-   AA          TAX
E94D-   20 29 E9    JSR   $E929         ; Namensvergleich Eintrag - Key Block
```

```
E950-  B0 0F      BCS   $E961       ; ungleich: NEXT
E952-  BD 11 D9   LDA   $D911,X     ; OPEN File im VCB-Eintrag?
E955-  30 14      BMI   $E96B       ; ja, Setzen von $FEB5/B6 + Exit
E957-  A9 00      LDA   #$00
E959-  9D 00 D9   STA   $D900,X     ; ansonsten wird der VCB-Eintrag
E95C-  9D 10 D9   STA   $D910,X       ungültig gemacht
E95F-  F0 08      BEQ   $E969       ; „always"
E961-  8A         TXA               ; nächster VCB-Eintrag
E962-  18         CLC
E963-  29 E0      AND   #$E0        ; X-Reg ist auch Index im Namen bei $DB1C,
E965-  69 20      ADC   #$20          zurück auf die Basisadresse
E967-  90 E3      BCC   $E94C       ; nächster VCB-Eintrag
E969-  18         CLC               ; ok. Nicht gefunden bzw. kein OPEN File
E96A-  60         RTS

E96B-  8D B5 FE   STA   $FEB5       ; OPEN File im „alten" VCB-Eintrag, die
E96E-  8E B6 FE   STX   $FEB6         Namen sind gleich, die Unitnummern
E971-  38         SEC                 stimmen aber nicht überein!
E972-  60         RTS
```

**– SPACE ON VOLUME?** (Berechnung der freien Blocks auf einem Volume)
Beim „Login" eines Volumes wird nur TOTAL BLOCKS im VCB gesetzt, die Anzahl der freien Blocks muß erst berechnet werden.

Xref: $F2C8 – WRITE – nach der Bestimmung der benötigten Blocks via SET MARK
$F97F – DESTROY – Aufruf mit „Blocks needed" = $00

```
E973-  AE 91 FE   LDX   $FE91       ; VCB-Index: Wenn die Berechnung bereits
E976-  BD 15 D9   LDA   $D915,X       seit dem „Login" durchgeführt wurde, sind
E979-  1D 14 D9   ORA   $D914,X       die Bytes $14/15 im VCB-Eintrag <> 0
E97C-  D0 52      BNE   $E9D0         gesetzt (Ausnahme: 00 Blocks frei).
```

Xref: $F79F – GET F'INFO für VolDIR, erzwingt Lesen und Auswertung der VBM

```
E97E-  20 22 EA   JSR   $EA22       ; berechnet die Maximalzahl der VBM-Blocks
E981-  8D 9C FE   STA   $FE9C         auf diesem Volume aus TOTAL BLOCKS
E984-  A9 00      LDA   #$00
E986-  8D 86 FE   STA   $FE86       ; $FE84/87: Zähler für freie Blocks
E989-  8D 87 FE   STA   $FE87
E98C-  A9 FF      LDA   #$FF
E98E-  8D 9B FE   STA   $FE9B       ; „no Space"-Flag und VBM „Extent"
E991-  20 93 EB   JSR   $EB93       ; REWRITE VBM – falls sich noch irgendein
E994-  B0 4E      BCS   $E9E4         geänderter VBM-Block im Speicher befindet
E996-  AE 91 FE   LDX   $FE91       ; VCB-Index
E999-  BD 1A D9   LDA   $D91A,X
E99C-  85 46      STA   $46         ; die Nummer des (ersten) VBM-Blocks
E99E-  BD 1B D9   LDA   $D91B,X       wird gesetzt
E9A1-  85 47      STA   $47
```

– Loop durch alle möglichen VBM-Blocks auf diesem Volume

```
E9A3-  20 EE EB   JSR   $EBEE       ; liest einen VBM-Block auf $DC00(!)
E9A6-  B0 3C      BCS   $E9E4
E9A8-  20 E5 E9   JSR   $E9E5       ; Anzahl der freien Blocks => $FE86/87
E9AB-  CE 9C FE   DEC   $FE9C       ; Herunterzähler (s. $DB74)
E9AE-  30 09      BMI   $E9B9       ; keine weiteren VBM-Blocks
```

```
E9B0-   E6 46       INC   $46
E9B2-   D0 EF       BNE   $E9A3        ; ansonsten nächster VBM-Block: Loop
E9B4-   E6 47       INC   $47
E9B6-   4C A3 E9    JMP   $E9A3

E9B9-   AE 91 FE    LDX   $FE91
E9BC-   AD 9B FE    LDA   $FE9B        ; mindestens ein freier Block gefunden?
E9BF-   30 20       BMI   $E9E1        ; nein - ERR=48
E9C1-   9D 1C D9    STA   $D91C,X      ; „VBM extent", auf dem ein freier Block
E9C4-   AD 87 FE    LDA   $FE87          gefunden wurde - $FE9B von $E9E5 gesetzt
E9C7-   9D 15 D9    STA   $D915,X      ; Anzahl der freien Blocks
E9CA-   AD 86 FE    LDA   $FE86
E9CD-   9D 14 D9    STA   $D914,X
```

– Bestimmung der freien Blocks ist erfolgt

```
E9D0-   BD 14 D9    LDA   $D914,X      ; Anzahl der freien Blocks
E9D3-   38          SEC
E9D4-   ED 94 FE    SBC   $FE94        ; BLOCKS NEEDED (low)
E9D7-   BD 15 D9    LDA   $D915,X
E9DA-   ED 95 FE    SBC   $FE95        ; hi - sehr gründlich & nie benutzt ($00)
E9DD-   90 02       BCC   $E9E1        ; VOLUME FULL
E9DF-   18          CLC
E9E0-   60          RTS

E9E1-   A9 48       LDA   #$48         ; ERR=48: VOLUME FULL
E9E3-   38          SEC
E9E4-   60          RTS
```

**– CALCFREE –**

Berechnung der freien Blocks in einem VBM-Block, setzt $FE9B = Extentno.

Xref: $E9A8 – Berechnung sämtlicher freien Blocks auf einem Volume

```
E9E5-   A0 00       LDY   #$00
E9E7-   B9 00 DC    LDA   $DC00,Y
E9EA-   F0 03       BEQ   $E9EF        ; alle 8 Blocks belegt
E9EC-   20 12 EA    JSR   $EA12        ; zählt die gesetzten Bits im Byte
E9EF-   B9 00 DD    LDA   $DD00,Y
E9F2-   F0 03       BEQ   $E9F7        ; „obere Hälfte" des VBM-Blocks
E9F4-   20 12 EA    JSR   $EA12
E9F7-   C8          INY
E9F8-   D0 ED       BNE   $E9E7
E9FA-   2C 9B FE    BIT   $FE9B        ; „No Space"-Flag bereits zurückgesetzt?
E9FD-   10 12       BPL   $EA11        ; RTS wenn ja
E9FF-   AD 86 FE    LDA   $FE86
EA02-   0D 87 FE    ORA   $FE87
EA05-   F0 0A       BEQ   $EA11        ; RTS wenn 00 00 Blocks frei
EA07-   20 22 EA    JSR   $EA22        ; nocheinmal Berechnung der max. VBM-Blocks
EA0A-   38          SEC
EA0B-   ED 9C FE    SBC   $FE9C        ; minus Herunterzähler
EA0E-   8D 9B FE    STA   $FE9B        ; = erster VBM-Block mit freien Blocks
EA11-   60          RTS                  (Extent No., d.h. 0,1,2..)
```

– Zählung der gesetzten Bits im Akku, Erhöhung von $FE86/87

Xref: $E9EC – CALCFREE Loop für untere Hälfte eines VBM-Blocks
$E9F4 – CALCFREE Loop für obere Hälfte eines VBM-Blocks

```
EA12-  0A          ASL
EA13-  90 08       BCC   $EA1D
EA15-  EE 86 FE    INC   $FE86        ; Bit gesetzt: Erhöhung des
EA18-  D0 03       BNE   $EA1D          „Free Space Count"
EA1A-  EE 87 FE    INC   $FE87
EA1D-  09 00       ORA   #$00         ; Loop, solange noch Bits gesetzt sind
EA1F-  D0 F1       BNE   $EA12
EA21-  60          RTS
```

– Berechnung der maximal möglichen Zahl der VBM-Blocks -1 für ein Volume

Xref: $E97E – SPACE ON VOL – Berechnung des Herunterzählers für den Loop
$EA07 – CALCFREE – Setzen des ersten VBM-Extent mit freien Blocks (=> $FE9B)

```
EA22-  AE 91 FE    LDX   $FE91        ; VCB-Index
EA25-  BC 13 D9    LDY   $D913,X      ; TOTAL BLOCKS (hi)
EA28-  BD 12 D9    LDA   $D912,X      ; lo – geben zusammen die Gesamtzahl
EA2B-  D0 01       BNE   $EA2E          der Blocks auf dem Volume an.
EA2D-  88          DEY
EA2E-  98          TYA                  z.B.: 10 00 Blocks => 00, aber
EA2F-  4A          LSR                        10 01 Blocks => 01, d.h. es
EA30-  4A          LSR                  existieren dann 2 VBM-Blocks
EA31-  4A          LSR
EA32-  4A          LSR                ; TOTAL BLOCKS HI / 16 = Anzahl der
EA33-  60          RTS                  existierenden VBM-Blocks -1
```

– **DEALLOCATE BLOCK** (in der VBM)

Xref: $F9D3 – DESTROY – den „höchstwertigen" Block erst nach Entfernen des Inhalts
$FA4C – DESTROY – Subdirectories Block per Block via Forward Reference
$FB9A – REMOVE BLOCKS – den vorher „höchstwertigen" Block, danach Typ = Typ-1
$FBD0 – REMOVE BLOCKS – den Inhalt eines Index/Masterblocks

```
EA34-  8E 9C FE    STX   $FE9C        ; Blocknummer hi
EA37-  48          PHA                ; Blocknummer lo
EA38-  AE 91 FE    LDX   $FE91        ; VCB-Index
EA3B-  BD 13 D9    LDA   $D913,X      ; TOTAL BLOCKS hi
EA3E-  CD 9C FE    CMP   $FE9C        ; Test: wird hier versucht, einen Block
EA41-  68          PLA                  „out of Range" zu entfernen?
EA42-  90 6E       BCC   $EAB2        ; wenn ja, FILE STRUCTURE DAMAGED
EA44-  AA          TAX                ; Blocknummer lo
EA45-  29 07       AND   #$07
EA47-  A8          TAY
EA48-  B9 00 FE    LDA   $FE00,Y      ; Bit Mask: $80/$40/$20/$10/$08/$04/$02/$01
EA4B-  8D 9B FE    STA   $FE9B        ; dieses Bit soll wieder gesetzt werden
EA4E-  8A          TXA                ; Reload Blocknummer lo
EA4F-  4E 9C FE    LSR   $FE9C
EA52-  6A          ROR
EA53-  4E 9C FE    LSR   $FE9C
EA56-  6A          ROR
EA57-  4E 9C FE    LSR   $FE9C        ; Blocknummer/8 => Basis-Index innerhalb
EA5A-  6A          ROR                  eines VBM-Blocks
```

```
EA5B-  8D A2 FE    STA   $FEA2
EA5E-  4E 9C FE    LSR   $FE9C        ; Blockno hi/16 => VBM „extent" Nummer
EA61-  2E A4 FE    ROL   $FEA4        ; = Speicherseite im VBM-Block (0 oder 1)
EA64-  20 64 EB    JSR   $EB64        ; „rettet" die VBM eines anderen Units,
EA67-  B0 48       BCS   $EAB1          falls noch im Speicher
EA69-  AD A9 FE    LDA   $FEA9        ; „Extent No." des momentanen VBM-Blocks
EA6C-  CD 9C FE    CMP   $FE9C        ; vs. „Extent", der benötigt wird
EA6F-  F0 16       BEQ   $EA87
EA71-  20 93 EB    JSR   $EB93        ; anderer Extent, u.U. vom selben Unit und
EA74-  B0 3B       BCS   $EAB1          nicht von $EB64 gerettet. Explizites WRITE
EA76-  AD 9C FE    LDA   $FE9C        ; Nummer des benötigten Extents
EA79-  AE 91 FE    LDX   $FE91        ; VCB-Index
EA7C-  9D 1C D9    STA   $D91C,X      ; Setzen des momentanen Extents
EA7F-  AD A6 FE    LDA   $FEA6        ; Unitnummer
EA82-  20 A4 EB    JSR   $EBA4        ; und Lesen des Extents (Blocknummer 00 06
EA85-  B0 2A       BCS   $EAB1          + Extent Nummer)
```

– benötigter VBM-Block (Extent) im Speicher

```
EA87-  AC A2 FE    LDY   $FEA2        ; (Byte)-Basis des freizugebenden Blocks
EA8A-  4E A4 FE    LSR   $FEA4        ; Speicherseite (0 oder 1)
EA8D-  AD 9B FE    LDA   $FE9B        ; Mask mit dem zu setzenden Bit
EA90-  90 08       BCC   $EA9A        ; Carry gesetzt: Speicherseite 1
EA92-  19 00 DB    ORA   $DB00,Y
EA95-  99 00 DB    STA   $DB00,Y      ; Setzen des Bits (Freigeben des Blocks)
EA98-  B0 06       BCS   $EAA0
EA9A-  19 00 DA    ORA   $DA00,Y
EA9D-  99 00 DA    STA   $DA00,Y
EAA0-  A9 80       LDA   #$80
EAA2-  0D A5 FE    ORA   $FEA5        ; Setzen des „VBM altered" Flag
EAA5-  8D A5 FE    STA   $FEA5
EAA8-  EE C2 FE    INC   $FEC2        ; für REMOVE BLOCKS/DESTROY: Zähler der
EAAB-  D0 03       BNE   $EAB0          freigegebenen Blocks
EAAD-  EE C3 FE    INC   $FEC3
EAB0-  18          CLC
EAB1-  60          RTS

EAB2-  A9 5A       LDA   #$5A         ; ERR#5A: FILE STRUCTURE DAMAGED
EAB4-  38          SEC
EAB5-  60          RTS
```

**– ALLOCATE NEXT BLOCK** – Suchen und Belegen des nächsten freien Blocks

Xref: $E341 – CREATE (Anhängen eines Blocks an eine Subdirectory)
$E469 – CREATE (Belegen eines Blocks für einen neuen File/Subdirectory)
$F438 – WRITE (Weiterer Datenblock bzw. Master/Index bei Typerweiterung)

```
EAB6-  20 64 EB    JSR   $EB64        ; Lesen des (ersten) VBM-Blocks auf $DA00
EAB9-  B0 23       BCS   $EADE          und Rewrite eventueller anderer VBM-Blocks
EABB-  A0 00       LDY   #$00
EABD-  8C A4 FE    STY   $FEA4        ; Speicherseite 0 im VBM-Block (Flag)
EAC0-  B9 00 DA    LDA   $DA00,Y
EAC3-  D0 1A       BNE   $EADF        ; freier Block gefunden
EAC5-  C8          INY
EAC6-  D0 F8       BNE   $EAC0        ; zuerst Absuchen der unteren Speicherseite
EAC8-  EE A4 FE    INC   $FEA4        ; => Speicherseite 1
EACB-  EE A3 FE    INC   $FEA3        ; Blocknummer / $800 (VBM-Block/2)
```

```
EACE-   B9 00 DB    LDA   $DB00,Y
EAD1-   D0 0C       BNE   $EADF         ; Absuchen der oberen Speicherseite
EAD3-   C8          INY
EAD4-   D0 F8       BNE   $EACE
EAD6-   EE A3 FE    INC   $FEA3
EAD9-   20 42 EB    JSR   $EB42         ; wenn weitere VBM-Blocks möglich, Laden
EADC-   90 DD       BCC   $EABB           des nächsten VBM-Blocks und Loop
EADE-   60          RTS                 ; mit ERR=48: VOLUME FULL (von $DD35)

EADF-   8C A2 FE    STY   $FEA2         ; freier Block gefunden: Index in der Seite
EAE2-   AD A3 FE    LDA   $FEA3         ; Blocknummer / $800
EAE5-   8D 87 FE    STA   $FE87
EAE8-   98          TYA                 ; 1 VBM-Byte = 8 Blocks, d.h. der Index
EAE9-   0A          ASL                   innerhalb einer Seite * 8 ergibt die
EAEA-   2E 87 FE    ROL   $FE87           Basis des freien Blocks
EAED-   0A          ASL                   in $FE87 (hi) und dem X-Reg (lo)
EAEE-   2E 87 FE    ROL   $FE87
EAF1-   0A          ASL
EAF2-   2E 87 FE    ROL   $FE87
EAF5-   AA          TAX                 ; X = $00/$08/$10/$18.....
EAF6-   38          SEC                 ; Marker im 8-Block-Byte
EAF7-   AD A4 FE    LDA   $FEA4         ; Speicherseite: 0 oder 1
EAFA-   F0 05       BEQ   $EB01
EAFC-   B9 00 DB    LDA   $DB00,Y
EAFF-   B0 03       BCS   $EB04
EB01-   B9 00 DA    LDA   $DA00,Y
EB04-   2A          ROL                 ; den Marker hinein
EB05-   B0 03       BCS   $EB0A
EB07-   E8          INX                 ; die Blocknummer wird erhöht (0..7)
EB08-   D0 FA       BNE   $EB04         ; Loop, bis ein gesetztes Byte „abfällt"
EB0A-   4A          LSR                 ; Shift & Loop rückwärts, bis der Marker
EB0B-   90 FD       BCC   $EB0A           wieder erscheint
EB0D-   8E 86 FE    STX   $FE86         ; Blocknummer lo
EB10-   AE A4 FE    LDX   $FEA4         ; Speicherseite in der VBM
EB13-   D0 05       BNE   $EB1A
EB15-   99 00 DA    STA   $DA00,Y       ; Zurück in die VBM. Das „abgefallene"
EB18-   F0 03       BEQ   $EB1D           Bit bleibt Null, d.h. der Block
EB1A-   99 00 DB    STA   $DB00,Y         ist jetzt belegt.
EB1D-   A9 80       LDA   #$80
EB1F-   0D A5 FE    ORA   $FEA5         ; „VBM altered" Flag wird gesetzt
EB22-   8D A5 FE    STA   $FEA5
EB25-   AC 91 FE    LDY   $FE91         ; VCB-Index für das Volume
EB28-   B9 14 D9    LDA   $D914,Y
EB2B-   E9 01       SBC   #$01          ; die Anzahl der freien Blocks auf dem
EB2D-   99 14 D9    STA   $D914,Y         Volume wird um eins erniedrigt
EB30-   B0 08       BCS   $EB3A
EB32-   B9 15 D9    LDA   $D915,Y
EB35-   E9 00       SBC   #$00
EB37-   99 15 D9    STA   $D915,Y
EB3A-   18          CLC
EB3B-   AD 86 FE    LDA   $FE86         ; Nummer des belegten Blocks lo
EB3E-   AC 87 FE    LDY   $FE87         ; hi
EB41-   60          RTS
```

– Test, ob mehrere VBM-Blocks auf dem Volume / ggf. Laden des nächsthöheren Extent

Xref: $EAD9 – ALLOCATE NEXT BLOCK – kein freier Block im momentanen VBM-Block

```
EB42-   AC 91 FE    LDY   $FE91         ; VCB-Index
EB45-   B9 13 D9    LDA   $D913,Y       ; TOTAL BLOCKS hi
EB48-   4A          LSR
```

```
EB49-   4A          LSR
EB4A-   4A          LSR
EB4B-   4A          LSR                 ; / 16 => Anzahl der VBM-Blocks
EB4C-   D9 1C D9    CMP   $D91C,Y       ; momentaner Extent
EB4F-   F0 0F       BEQ   $EB60         ; bereits erreicht (letzter VBM-Block)?
EB51-   B9 1C D9    LDA   $D91C,Y
EB54-   18          CLC
EB55-   69 01       ADC   #$01          ; Setzen des nächsten Extents
EB57-   99 1C D9    STA   $D91C,Y
EB5A-   20 93 EB    JSR   $EB93         ; Zurückschreiben des alten Extents
EB5D-   4C 64 EB    JMP   $EB64         ; RTS via Lesen des neuen Extents
EB60-   A9 48       LDA   #$48
EB62-   38          SEC                 ; letzter VBM-Block bereits erreicht:
EB63-   60          RTS                   ERR#48: VOLUME FULL
```

– **GET VBM** – Lesen eines VBM-Blocks auf $DA00/$DB00
VBM-Blocks anderer Units werden zurückgeschrieben, wenn das Flag „VBM altered" gesetzt ist. RTS, wenn ein VBM-Block (beliebiger Extent!) des gewünschten Units bereits im Speicher ist.

Xref: $EA64 – DEALLOCATE BLOCK
$EAB6 – ALLOCATE NEXT BLOCK (Laden des ersten VBM-Blocks)
JP $EB5D – GET NEXT VBM BLOCK (s. oben)
$FA12 – DESTROY (Subdirectories)

```
EB64-   AC 91 FE    LDY   $FE91        ; VCB-Index
EB67-   B9 10 D9    LDA   $D910,Y      ; Unitnummer
EB6A-   CD A6 FE    CMP   $FEA6        ; Unitnummer des momentanen VBM-Blocks
EB6D-   F0 0E       BEQ   $EB7D
EB6F-   20 93 EB    JSR   $EB93        ; ungleich: Rewrite des VBM-Blocks eines
EB72-   B0 1E       BCS   $EB92          anderen Units, wenn verändert
EB74-   AC 91 FE    LDY   $FE91
EB77-   B9 10 D9    LDA   $D910,Y      ; Setzen der neuen Unitnummer
EB7A-   8D A6 FE    STA   $FEA6        ; als „VBM-Block momentan im Speicher"

EB7D-   AC A5 FE    LDY   $FEA5        ; „VBM altered" Flag, via $EB6F nie gesetzt
EB80-   30 05       BMI   $EB87        ; VBM-Block verändert, nicht überschreiben!
EB82-   20 A4 EB    JSR   $EBA4        ; sonst Lesen des neuen VBM-Blocks
EB85-   B0 0B       BCS   $EB92
EB87-   AC 91 FE    LDY   $FE91
EB8A-   B9 1C D9    LDA   $D91C,Y      ; VBM Extent
EB8D-   0A          ASL
EB8E-   8D A3 FE    STA   $FEA3        ; VBM-Extent * 2 = Blocknummer / $800
EB91-   18          CLC
EB92-   60          RTS
```

– **REWRITE** des alten VBM-Blocks, wenn verändert

Xref: $E49A/$E991/$EA71/$EB5A/$EB6F/$EE8E/$EEA8/$F5F0/$F9E8

```
EB93-   18          CLC
EB94-   AD A5 FE    LDA   $FEA5        ; „VBM altered"-Flag
EB97-   10 F9       BPL   $EB92        ; nicht verändert: RTS
EB99-   20 E6 EB    JSR   $EBE6        ; sonst Zurückschreiben auf das
EB9C-   B0 F4       BCS   $EB92          entsprechende Unit
```

```
EB9E-   A9 00       LDA   #$00          ; und Reset des „VBM altered"-Flags
EBA0-   8D A5 FE    STA   $FEA5
EBA3-   60          RTS
```

– **READ VBM** (auf $DA00/$DB00)

Xref: $EA82 – DEALLOCATE BLOCK
$EB82 – GET VBM

```
EBA4-   8D A6 FE    STA   $FEA6          ; Aufruf mit Unitnummer im Akku
EBA7-   AC 91 FE    LDY   $FE91          ; VCB-Index
EBAA-   B9 1C D9    LDA   $D91C,Y        ; VBM-Extentnummer (0, 1, 2 ...)
EBAD-   8D A9 FE    STA   $FEA9
EBB0-   18          CLC
EBB1-   79 1A D9    ADC   $D91A,Y        ; Nummer des ersten VBM-Blocks
EBB4-   8D A7 FE    STA   $FEA7
EBB7-   B9 1B D9    LDA   $D91B,Y
EBBA-   69 00       ADC   #$00
EBBC-   8D A8 FE    STA   $FEA8          ; $FEA7/A8: Nummer des mom. VBM-Blocks
EBBF-   A9 01       LDA   #$01           ; Befehl: READ
```

– Einsprung für WRITE von $EBE8

```
EBC1-   85 42       STA   $42            ; COMMAND für die RWTB
EBC3-   AD 30 BF    LDA   $BF30          ; die Nummer des zuletzt benutzten Units
EBC6-   48          PHA                    wird zwischengespeichert
EBC7-   AD A6 FE    LDA   $FEA6          ; die Unitnummer des VBM-Blocks
EBCA-   8D 30 BF    STA   $BF30            wird gesetzt
EBCD-   AD A7 FE    LDA   $FEA7
EBD0-   85 46       STA   $46            ; Blocknummer für die RWTB
EBD2-   AD A8 FE    LDA   $FEA8
EBD5-   85 47       STA   $47
EBD7-   AD 9C EA    LDA   $EA9C          ; = $DA, d.h. Ziel/Quelladresse = $DA00
EBDA-   20 F5 EB    JSR   $EBF5          ; Teil von R/W Block von/nach $DA00/$DC00
EBDD-   AA          TAX                  ; eventuelle ERR# von der RWTB
EBDE-   68          PLA
EBDF-   8D 30 BF    STA   $BF30          ; Restore der alten Unitnummer
EBE2-   90 01       BCC   $EBE5
EBE4-   8A          TXA                  ; eventuelle ERR#
EBE5-   60          RTS
```

– **WRITE VBM-Block** von $DA00

Xref: $EB99 – Rewrite eines geänderten VBM-Blocks

```
EBE6-   A9 02       LDA   #$02           ; RWTB-Befehl: WRITE
EBE8-   D0 D7       BNE   $EBC1          ; „always"
```

– **WRITE BLOCK** von $DC00/$DD00

Xref: $E354/$E373/$E3AC/$E488/$E501/$E526/$E564/
$F948/$FA38/$FAFF/$FB33/$FB82

```
EBEA-   A9 02       LDA   #$02           ; RWTB-Befehl: WRITE
EBEC-   D0 02       BNE   $EBF0          ; „always"
```

**– READ BLOCK** nach $DC00/$DD00

Xref: $E29B/$E382/$E4DF/$E510/$E54A/$E57E/$E628/
$E67A/$E7F5/$E854/$E9A3$F579/$F595/$F927/
$FA21/$FA5B/$FAE2/$FB18/$FB4C/$FB91

```
EBEE-   A9 01       LDA   #$01          ; RWTB-Befehl: READ

EBF0-   85 42       STA   $42           ; COMMAND für die RWTB
EBF2-   AD 35 E6    LDA   $E635         ; = $DC, d.h. Ziel/Quelladresse = $DC00
```

Xref: $EBDA – READ VBM: Zieladresse ist $DA00/$DB00!

```
EBF5-   08          PHP                 ; Save des INT-Status
EBF6-   78          SEI
EBF7-   85 45       STA   $45           ; Ziel/Quelladresse hi
EBF9-   A9 00       LDA   #$00
EBFB-   85 44       STA   $44           ; Ziel/Quelladresse lo
EBFD-   8D 0F BF    STA   $BF0F         ; SYSERR auf #00
EC00-   A9 FF       LDA   #$FF
EC02-   8D B2 FE    STA   $FEB2         ; „Mystery Flag", s. READ Stufe 3
EC05-   AD 30 BF    LDA   $BF30         ; Unitnummer
EC08-   85 43       STA   $43
EC0A-   20 DA DE    JSR   $DEDA         ; DO DISK I/O
EC0D-   B0 03       BCS   $EC12
EC0F-   28          PLP                 ; Restore des INT-Status
EC10-   18          CLC
EC11-   60          RTS
EC12-   28          PLP                 ; Error Exit: Restore des INT-Status
EC13-   38          SEC
EC14-   60          RTS
```

**$CF – GET MARK COMMAND** ; File muß OPEN sein – Set File Data vorher

```
EC15-   AE 92 FE    LDX   $FE92         ; FCB-Index, gesetzt von Set File Data
EC18-   A0 02       LDY   #$02
EC1A-   BD 12 D8    LDA   $D812,X       ; die momentane Position im File (3 Bytes)
EC1D-   91 40       STA   ($40),Y         wird in den „user"-Buffer kopiert
EC1F-   E8          INX
EC20-   C8          INY
EC21-   C0 05       CPY   #$05
EC23-   D0 F5       BNE   $EC1A
EC25-   18          CLC
EC26-   60          RTS                 ; => $DE78
```

– Error Exit für SET MARK (nur für „user"-Command)

```
EC27-   A9 4D       LDA   #$4D          ; ERR#4D: POSITION OUT OF RANGE
EC29-   38          SEC
EC2A-   60          RTS
```

**$CE – SET MARK COMMAND** ; File muß OPEN sein – Set File Data vorher

```
EC2B-   A0 04       LDY   #$04
EC2D-   AE 92 FE    LDX   $FE92         ; FCB-Index, gesetzt von Set File Data
EC30-   E8          INX
```

```
EC31-   E8          INX
EC32-   38          SEC                 ; Marker für folgendes CMP
EC33-   B1 40       LDA   ($40),Y
EC35-   99 A8 FE    STA   $FEA8,Y       ; die „Position gewünscht" wird aus dem
EC38-   90 08       BCC   $EC42           „user"-Buffer kopiert und gleichzeitig
EC3A-   DD 15 D8    CMP   $D815,X       ; mit dem EOF verglichen
EC3D-   90 03       BCC   $EC42         ; Sprung wenn „Position gewünscht" < EOF
EC3F-   D0 E6       BNE   $EC27         ; ERR#4D wenn „Position gewünscht" > EOF
EC41-   CA          DEX
EC42-   88          DEY
EC43-   98          TYA
EC44-   49 01       EOR   #$01          ; das Carry beibehalten!
EC46-   D0 EB       BNE   $EC33
```

**– POSITION (SET MARK)** – Einsprung für alle internen Aufrufe

Xref: $EF78 – OPEN: „Position jetzt" auf FF 00 00, „gewünscht" auf 00 00 00
=> erzwingt das Lesen des ersten Blocks in den Filepuffer
$F055 – READ: Byte-per-Byte – nächster Block wird benötigt
$F07F – READ: Block-per-Block – nächster Block ins Zielgebiet (Pointer!)
JP $F10D – READ/WRITE gemeinsamer Exit:
READ: nach Stufe 2/3 Reload des Filepuffers
WRITE: wenn an Blockgrenze, wird der Block geschrieben
$F1BB – READ: für Directories – nächster Block
$F2B0 – WRITE: nächster Block des Files – (Typ)Erweiterungsbits werden gesetzt, wenn File-Länge überschritten wird
$F743 – SET EOF: Erzwingt Reload des Filepuffers nach Verkürzung des Files, ebenfalls durch „Verdrehung" der „Position jetzt"

```
EC48-   AC 92 FE    LDY   $FE92         ; FCB-Index
EC4B-   B9 13 D8    LDA   $D813,Y       ; „mittlere" Position jetzt = Blockno * 2
EC4E-   29 FE       AND   #$FE
EC50-   8D 86 FE    STA   $FE86
EC53-   AD AB FE    LDA   $FEAB         ; „Position gewünscht" – mittleres Byte
EC56-   38          SEC
EC57-   ED 86 FE    SBC   $FE86
EC5A-   8D 86 FE    STA   $FE86         ; = Differenz in Blocks * 2
EC5D-   90 0F       BCC   $EC6E         ; negativ („rückwärts")
EC5F-   C9 02       CMP   #$02
EC61-   B0 0B       BCS   $EC6E         ; positiv > 2, d.h. > 1 Block
EC63-   AD AC FE    LDA   $FEAC
EC66-   D9 14 D8    CMP   $D814,Y       ; Hibytes „jetzt/gewünscht"
EC69-   D0 03       BNE   $EC6E
EC6B-   4C 89 ED    JMP   $ED89         ; ok. Block bereits im Speicher

EC6E-   B9 07 D8    LDA   $D807,Y       ; Storage Type
EC71-   F0 07       BEQ   $EC7A         ; Sprung wenn $00 (deleted)
EC73-   C9 04       CMP   #$04
EC75-   90 0C       BCC   $EC83         ; ist Datenfile
EC77-   4C BB ED    JMP   $EDBB         ; SET MARK für (Sub)diretories

EC7A-   A0 92       LDY   #$92    ****  ; Programmierfehler (s. „Spezielles")
EC7C-   99 00 D8    STA   $D800,Y
EC7F-   A9 43       LDA   #$43          ; ERR#43: INVALID REFERENCE NUMBER
EC81-   38          SEC
EC82-   60          RTS
```

– Positionierung für die Typen 1/2/3

```
EC83-   B9 07 D8    LDA   $D807,Y      ; Storage Type
EC86-   8D 96 FE    STA   $FE96
EC89-   B9 08 D8    LDA   $D808,Y      ; STATUS im FCB-Eintrag
EC8C-   29 40       AND   #$40         ; geänderter Datenblock im Speicher?
EC8E-   F0 05       BEQ   $EC95
EC90-   20 94 EE    JSR   $EE94        ; ja, Zurückschreiben
EC93-   B0 69       BCS   $ECFE
EC95-   AC 92 FE    LDY   $FE92        ; FCB-Index
EC98-   B9 14 D8    LDA   $D814,Y      ; MSByte der „Position jetzt"
EC9B-   29 FE       AND   #$FE
EC9D-   8D 86 FE    STA   $FE86
ECA0-   AD AC FE    LDA   $FEAC        ; MSByte „Position gewünscht"
ECA3-   38          SEC                ; wenn die Differenz negativ oder > 2
ECA4-   ED 86 FE    SBC   $FE86          ist, wird ein anderer Indexblock benötigt
ECA7-   90 1E       BCC   $ECC7        ; negativ
ECA9-   C9 02       CMP   #$02
ECAB-   B0 1A       BCS   $ECC7        ; > 2
ECAD-   AE 96 FE    LDX   $FE96        ; Positionierung im selben Indexblock:
ECB0-   CA          DEX                ; hat der File einen Storage Type >1?
ECB1-   D0 7A       BNE   $ED2D        ; Ja, Sprung
```

– Laden des Datenblocks für Type 1 bzw. Typerweiterung

```
ECB3-   AD AB FE    LDA   $FEAB        ; „mittleres Byte" von „Position gewünscht"
ECB6-   4A          LSR
ECB7-   0D AC FE    ORA   $FEAC        ; MSByte „Position gewünscht"
ECBA-   D0 5F       BNE   $ED1B        ; Erweiterung, wenn „gewünscht" > 00 01 FF
ECBC-   B9 0C D8    LDA   $D80C,Y      ; „Position gewünscht" < 00 02 00: Bytes
ECBF-   85 46       STA   $46            $0C/0D enthalten die „Nummer erster Block"
ECC1-   B9 0D D8    LDA   $D80D,Y        d.h. die Nummer des Datenblocks
ECC4-   4C 7F ED    JMP   $ED7F        ; Lesen => BLOCK IN MEMORY
```

– Differenz „jetzt" und „gewünscht" > 256 Blocks. Neuer Indexblock benötigt.
Ist die „Position gewünscht" innerhalb des zulässigen Bereichs, wird der Storage Type nicht heraufgesetzt. Wird von OPEN und SET EOF benutzt, um ein Lesen in den Filepuffer zu erzwingen.

```
ECC7-   B9 08 D8    LDA   $D808,Y      ; STATUS
ECCA-   29 80       AND   #$80         ; geänderter Indexblock im Speicher?
ECCC-   F0 05       BEQ   $ECD3
ECCE-   20 A8 EE    JSR   $EEA8        ; ja, Zurückschreiben
ECD1-   B0 2B       BCS   $ECFE
ECD3-   AE 96 FE    LDX   $FE96        ; Storage Type
ECD6-   E0 03       CPX   #$03
ECD8-   F0 26       BEQ   $ED00        ; Type 3, keine Erweiterung
ECDA-   AD AC FE    LDA   $FEAC        ; MSByte „Position gewünscht"
ECDD-   4A          LSR
ECDE-   08          PHP                ; „Push Z-Flag"
ECDF-   A9 07       LDA   #$07         ; Erweiterungsbits: 1 Master/1 Index/1 Data
ECE1-   28          PLP
ECE2-   D0 62       BNE   $ED46        ; MSByte „gewünscht" > 1 => Erweiterung
ECE4-   20 AF ED    JSR   $EDAF        ; im Bereich Typ 2: Erweiterungsbits löschen
ECE7-   CA          DEX
ECE8-   F0 C9       BEQ   $ECB3        ; File hat den Typ 1: Test auf Bereich
ECEA-   20 3B EE    JSR   $EE3B        ; Type 2: Indexblock („erster Block") lesen
ECED-   B0 0F       BCS   $ECFE
ECEF-   AC 92 FE    LDY   $FE92        ; FCB-Index
ECF2-   A5 46       LDA   $46
ECF4-   99 0E D8    STA   $D80E,Y      ; Setzen der „Nummer Indexblock im Speicher"
ECF7-   A5 47       LDA   $47            im FCB-Eintrag
```

```
ECF9-   99 0F D8    STA   $D80F,Y
ECFC-   90 2F       BCC   $ED2D        ; von $ECF9 her ist das ein „always"
ECFE-   38          SEC
ECFF-   60          RTS
```

– Type 3: Lesen des Master-Blocks

```
ED00-   20 AF ED    JSR   $EDAF        ; Erweiterungsbits in STATUS löschen
ED03-   20 3B EE    JSR   $EE3B        ; Lesen des Masterblocks („erster Block")
ED06-   B0 F6       BCS   $ECFE          in den Indexteil des Filepuffers
ED08-   AD AC FE    LDA   $FEAC        ; MSByte „Position gewünscht"
ED0B-   4A          LSR                ; / 2 => Position des Indexblocks im Master
ED0C-   A8          TAY
ED0D-   B1 48       LDA   ($48),Y      ; Hat dieser Indexblock die Nummer 00 00,
ED0F-   E6 49       INC   $49            d.h. existiert er (noch) nicht, wird die
ED11-   D1 48       CMP   ($48),Y        Typerweiterung benutzt, um einen neuen
ED13-   D0 0B       BNE   $ED20          Indexblock zu erstellen
ED15-   C9 00       CMP   #$00
ED17-   D0 07       BNE   $ED20        ; Indexblock existiert, Sprung
ED19-   C6 49       DEC   $49
```

– neuer Indexblock benötigt (Type 3 und Erweiterung Type 1 => Type 2)

```
ED1B-   A9 03       LDA   #$03         ; Erweiterungsbits: 1 Index/ 1 Datenblock
ED1D-   4C 46 ED    JMP   $ED46
```

– Type 3: Setzen der Indexblocknummer

```
ED20-   85 46       STA   $46          ; Setzen der Nummer des Indexblocks
ED22-   B1 48       LDA   ($48),Y
ED24-   85 47       STA   $47
ED26-   C6 49       DEC   $49
ED28-   20 1D EE    JSR   $EE1D        ; Lesen des Indexblocks
ED2B-   B0 D1       BCS   $ECFE
```

– Indexblock im Speicher (Typen 2 und 3)

```
ED2D-   AD AC FE    LDA   $FEAC        ; MSByte „Position gewünscht"
ED30-   4A          LSR
ED31-   AD AB FE    LDA   $FEAB        ; mittleres Byte
ED34-   6A          ROR                ; / 2 => Position des Datenblocks im Index
ED35-   A8          TAY
ED36-   B1 48       LDA   ($48),Y      ; derselbe Test wie beim Masterblock: hat
ED38-   E6 49       INC   $49            der Datenblock die Nummer 00 00,
ED3A-   D1 48       CMP   ($48),Y        dann existiert er noch nicht und muß
ED3C-   D0 3B       BNE   $ED79          via Erweiterung erstellt werden
ED3E-   C9 00       CMP   #$00
ED40-   D0 37       BNE   $ED79        ; ok. Datenblock existiert
ED42-   A9 01       LDA   #$01         ; Erweiterungsbit 0: 1 Datenblock
ED44-   C6 49       DEC   $49
```

– Erweiterung bzw. Erstellung eines neuen Index/Datenblocks: Je nach gesetzten Erweiterungsbits wird der Datenteil bzw. der Indexteil des Filepuffers gelöscht und dient als neuer (nächster) Block.

```
ED46-   AC 92 FE    LDY   $FE92        ; FCB-Index
ED49-   19 08 D8    ORA   $D808,Y      ; Einsprung mit Erweiterungsbits im Akku
ED4C-   99 08 D8    STA   $D808,Y      ; => Erweiterungsbits in STATUS
ED4F-   4A          LSR
ED50-   4A          LSR
ED51-   20 67 ED    JSR   $ED67        ; Löschen des Datenteils
```

```
ED54-   90 33       BCC   $ED89          ; Bit 1 nicht gesetzt: kein neuer Indexblock
ED56-   91 48       STA   ($48),Y
ED58-   C8          INY                  ; Löschen des Indexteils im Filepuffer
ED59-   D0 FB       BNE   $ED56
ED5B-   E6 49       INC   $49
ED5D-   91 48       STA   ($48),Y
ED5F-   C8          INY
ED60-   D0 FB       BNE   $ED5D
ED62-   C6 49       DEC   $49
ED64-   4C 89 ED    JMP   $ED89          ; BLOCK IN MEMORY (hier leerer Block)
```

– Löschung des Datenteils im Filepuffer

Xref: $ED51 – Datenblock existiert (noch) nicht
$F1D5 – READ für Directories: EOF überschritten

```
ED67-   A9 00       LDA   #$00
ED69-   A8          TAY
ED6A-   91 4A       STA   ($4A),Y        ; $004A/4B zeigen auf den Datenteil des
ED6C-   C8          INY                    Filepuffers
ED6D-   D0 FB       BNE   $ED6A
ED6F-   E6 4B       INC   $4B
ED71-   91 4A       STA   ($4A),Y
ED73-   C8          INY
ED74-   D0 FB       BNE   $ED71
ED76-   C6 4B       DEC   $4B
ED78-   60          RTS
```

– Holen der Datenblocknummer aus dem gelesenen Indexblock

```
ED79-   85 46       STA   $46           ; Einsprung mit Blocknummer low im Akku
ED7B-   B1 48       LDA   ($48),Y       ; und gesetztem Index: Blocknummer hi
ED7D-   C6 49       DEC   $49
```

–Lesen des Datenblocks, weiter mit BLOCK IN MEMORY

Xref: $EC04- Blocknummer für Type 1 gesetzt

```
ED7F-   85 47       STA   $47           ; Einsprung mit Blocknummer hi im Akku
ED81-   20 04 EE    JSR   $EE04         ; Lesen des Blocks in den Datenteil
ED84-   B0 28       BCS   $EDAE           des Filepuffers
ED86-   20 AF ED    JSR   $EDAF         ; Reset der Erweiterungsbits in STATUS
```

**– BLOCK IN MEMORY**

Xref: $EC6B – JP vom SET MARK Entry – Datenblock ist bereits im Speicher
$ED64 – JP von SET MARK – nach Löschen der entsprechenden Filepuffer-Teile
$F1D2 – READ für Directories – EOF überschritten, Setzen der Position im FCB

```
ED89-   AC 92 FE    LDY   $FE92         ; FCB-Index
ED8C-   C8          INY
ED8D-   C8          INY
ED8E-   A2 02       LDX   #$02
```

```
ED90-   B9 12 D8    LDA   $D812,Y      ; Kopieren der alten „Position jetzt"
ED93-   9D 8D FE    STA   $FE8D,X        (nur für den Error Exit von WRITE)
ED96-   BD AA FE    LDA   $FEAA,X
ED99-   99 12 D8    STA   $D812,Y      ; neue „Position jetzt" in den FCB-Eintrag
ED9C-   88          DEY
ED9D-   CA          DEX
ED9E-   10 F0       BPL   $ED90
EDA0-   18          CLC
EDA1-   A5 4A       LDA   $4A          ; lo-Byte des Datenpointers im Filepuffer
EDA3-   85 4C       STA   $4C          ; = $00
EDA5-   AD AB FE    LDA   $FEAB
EDA8-   29 01       AND   #$01         ; $004C/4D auf eine Seite innerhalb des
EDAA-   65 4B       ADC   $4B            Datenteils im Filepuffer
EDAC-   85 4D       STA   $4D
EDAE-   60          RTS
```

– Reset der Erweiterungsbits in STATUS – keine neuen Blocks/Erweiterung benötigt

Xref: $ECE4 – SET MARK: „neuer" Indexblock für Type 2 – Erzwingung durch OPEN/SET EOF
$ED00 – SET MARK: File hat bereits Type 3
$ED86 – SET MARK: „Position gewünscht" ist im Bereich des momentanen Types
$F727 – SET EOF: File wurde verkürzt – Erzwingung eines Reads

```
EDAF-   AC 92 FE    LDY   $FE92        ; FCB-Index
EDB2-   B9 08 D8    LDA   $D808,Y      ; STATUS
EDB5-   29 F8       AND   #$F8         ; Erweiterungsbits: Bit 0/1/2
EDB7-   99 08 D8    STA   $D808,Y
EDBA-   60          RTS
```

**– SET MARK IN DIRECTORY FILES**

Xref: $EC77 – SET MARK Entry: File hat nicht den Type 0/1/2/3

```
EDBB-   C9 0D       CMP   #$0D         ; Subdirectory oder VolDIR (s. GET FILE –
EDBD-   F0 05       BEQ   $EDC4          „Fälschung" des Storage Types)
EDBF-   A9 4A       LDA   #$4A         ; ERR#4A: INCOMPATIBLE FILE FORMAT
EDC1-   20 09 BF    JSR   $BF09        ; SET SYSERR; RTS z. Caller von SET MARK
EDC4-   AD 86 FE    LDA   $FE86        ; Differenz „jetzt – gewünscht" in Blocks
EDC7-   4A          LSR                  * 2 => Teilung durch 2 ergibt
EDC8-   8D 9A FE    STA   $FE9A        ; den Herunterzähler
EDCB-   B9 13 D8    LDA   $D813,Y      ; mittleres Byte „Position jetzt"
EDCE-   CD AB FE    CMP   $FEAB        ; mittleres Byte „Position gewünscht"
EDD1-   90 0E       BCC   $EDE1        ; Sprung, wenn „jetzt" < „gewünscht"

EDD3-   A0 00       LDY   #$00         ; Positionierung „rückwärts": Index zur
EDD5-   20 EF ED    JSR   $EDEF          Backward Reference
EDD8-   B0 24       BCS   $EDFE
EDDA-   EE 9A FE    INC   $FE9A        ; und Heraufzählen der (negativen) Differenz
EDDD-   10 F4       BPL   $EDD3        ; => BLOCK IN MEMORY
EDDF-   30 A8       BMI   $ED89

EDE1-   A0 02       LDY   #$02         ; Index zur Forward Reference
EDE3-   20 EF ED    JSR   $EDEF        ; Lesen des nächsten Blocks
EDE6-   B0 16       BCS   $EDFE
EDE8-   CE 9A FE    DEC   $FE9A        ; Herunterzählen der (positiven) Differenz
```

```
EDEB-   D0 F4       BNE   $EDE1
EDED-   F0 9A       BEQ   $ED89       ; => BLOCK IN MEMORY
```

– Lesen des nächsten DIR-Blocks via Forward/Backward Reference

Xref: $EDD5/$EDE3 – s.o.

```
EDEF-   B1 4A       LDA   ($4A),Y     ; $004A/4B: Pointer zum Datenteil des
EDF1-   85 46       STA   $46           Filepuffers, in dem bereits ein
EDF3-   C8          INY                 Block der Directory (via OPEN) steht
EDF4-   D1 4A       CMP   ($4A),Y
EDF6-   D0 08       BNE   $EE00
EDF8-   C9 00       CMP   #$00
EDFA-   D0 04       BNE   $EE00       ; Reference existiert
EDFC-   A9 4C       LDA   #$4C        ; Ref = 00 00, d.h. Anfang/Ende überschritten
EDFE-   38          SEC                 ERR#4C: EOF ENCOUNTERED
EDFF-   60          RTS
EE00-   B1 4A       LDA   ($4A),Y
EE02-   85 47       STA   $47         ; Nummer des nächsten DIR-Blocks
```

–**READ BLOCK** in den Datenteil des Filepuffers

Xref: $ED81 – SET MARK: Blocknummer gesetzt, danach BLOCK IN MEMORY
$EF94 – OPEN für Directory-Files – Read des Key Blocks (SET MARK kann dafür nicht benutzt werden, s. $EDBB)

```
EE04-   A9 01       LDA   #$01        ; Befehl: READ
EE06-   85 42       STA   $42
EE08-   A2 4A       LDX   #$4A        ; => $004A/4B: Datenteil des Filepuffers
EE0A-   20 61 EE    JSR   $EE61       ; setzt die Zieladresse und liest den Block
EE0D-   B0 0D       BCS   $EE1C
EE0F-   AC 92 FE    LDY   $FE92       ; FCB-Index
EE12-   A5 46       LDA   $46
EE14-   99 10 D8    STA   $D810,Y     ; Setzen von „Nummer Datenblock im Speicher"
EE17-   A5 47       LDA   $47           im FCB-Eintrag
EE19-   99 11 D8    STA   $D811,Y
EE1C-   60          RTS
```

– **BLOCKREAD/WRITE** innerhalb des Filepuffers

Xref: $ED28 – SET MARK (Lesen des Indexblocks)

```
EE1D-   A9 01       LDA   #$01        ; Befehl: READ
EE1F-   85 42       STA   $42
EE21-   A2 48       LDX   #$48        ; => $0048/49: Indexteil des Filepuffers
EE23-   20 61 EE    JSR   $EE61       ; setzt Zieladresse und liest den Block
EE26-   B0 0E       BCS   $EE36
EE28-   AC 92 FE    LDY   $FE92       ; FCB-Index
EE2B-   A5 46       LDA   $46
EE2D-   99 0E D8    STA   $D80E,Y     ; Setzen der Nummer „Indexblock im Speicher"
EE30-   A5 47       LDA   $47           im FCB-Eintrag
EE32-   99 0F D8    STA   $D80F,Y
EE35-   18          CLC
EE36-   60          RTS
```

Xref: $EE91 – JP nach Rewrite eines VBM-Blocks. $EE8E wird aufgerufen von:
$F3CD – WRITE: Rewrite eines (neuen) Masterblocks
$F421 – WRITE: Typ-Erweiterung

```
EE37-  A9 02      LDA   #$02          ; Befehl: WRITE
EE39-  D0 02      BNE   $EE3D         ; „always"
```

**– READ INDEX/MASTER BLOCK** in den Indexteil des Filepuffers

Xref: $ECEA – SET MARK: File hat Type 2, Lesen des Indexblocks
$ED03 – SET MARK: File hat Type 3, Lesen des Masterblocks
$F3EB – WRITE: Typ-Erweiterung. Wenn bereits Type 2, Lesen des Indexblocks

```
EE3B-  A9 01      LDA   #$01          ; Befehl: READ

EE3D-  48         PHA
EE3E-  A9 0C      LDA   #$0C
EE40-  0D 92 FE   ORA   $FE92         ; FCB-Index
EE43-  A8         TAY                 ; => FCB-Bytes $0C/0D = „erster Block"
EE44-  68         PLA                 ; COMMAND
EE45-  A2 48      LDX   #$48          ; => $0048/49: Indexteil des Filepuffers
```

Xref: $EE9E – (Re)write des momentanen Datenblocks (Blockno von „$D80E/0F")
$EEB5 – (Re)write des momentanen Indexblocks (Blockno von „$D810/11")

```
EE47-  85 42      STA   $42           ; COMMAND für die RWTB
EE49-  B9 00 D8   LDA   $D800,Y       ; Y = FCB-Index + $0C/0E oder $10
EE4C-  85 46      STA   $46           ; Blocknummer lo
EE4E-  D9 01 D8   CMP   $D801,Y       ; Blocknummer hi
EE51-  D0 09      BNE   $EE5C         ; Test auf Blocknummer 00 00: wenn der
EE53-  C9 00      CMP   #$00            FCB als Index/Daten oder "erster Block"
EE55-  D0 05      BNE   $EE5C           keinen Eintrag enthält:
EE57-  A9 0C      LDA   #$0C
EE59-  20 0C BF   JSR   $BF0C         ; SYSTEM DEATH mit ERR=0C

EE5C-  B9 01 D8   LDA   $D801,Y
EE5F-  85 47      STA   $47           ; Blocknummer hi
```

Xref: $EE0A – READ BLOCK in den Datenteil des Filepuffers; X = $4A
$EE23 – READ BLOCK (Master/beliebiger Indexblock) in den Indexteil; X = $48

```
EE61-  08         PHP
EE62-  78         SEI                 ; Save des INT-Status und „disable"
EE63-  B5 00      LDA   $00,X         ; X = $48 (Index) oder $4A (Datenteil)
EE65-  85 44      STA   $44
EE67-  B5 01      LDA   $01,X         ; $0044/45: Ziel/Quelladresse für die RWTB
EE69-  85 45      STA   $45
EE6B-  AC 92 FE   LDY   $FE92         ; Unitnummer
EE6E-  B9 01 D8   LDA   $D801,Y
EE71-  8D 30 BF   STA   $BF30
EE74-  A9 FF      LDA   #$FF
EE76-  8D B2 FE   STA   $FEB2         ; „Mystery Flag", s. READ Stufe 3
```

```
EE79-  AD 30 BF    LDA   $BF30
EE7C-  85 43       STA   $43          ; Unitnummer für die RWTB
EE7E-  A9 00       LDA   #$00
EE80-  8D 0F BF    STA   $BF0F        ; SYSERR auf #00
EE83-  20 DA DE    JSR   $DEDA        ; DO DISK I/O
EE86-  B0 03       BCS   $EE8B
EE88-  28          PLP                ; Restore des INT-Status
EE89-  18          CLC
EE8A-  60          RTS
EE8B-  28          PLP                ; Error Exit: Restore des INT-Status
EE8C-  38          SEC
EE8D-  60          RTS
```

Xref: $F3CD/$F421 – s. $EE37

```
EE8E-  20 93 EB    JSR   $EB93        ; Rewrite VBM-Block, wenn verändert
EE91-  4C 37 EE    JMP   $EE37
```

### –(RE)WRITE CURRENT DATA BLOCK

Xref: $EC90 – SET MARK: Rewrite eines veränderten Datenblocks vor neuem Read
$F54A – FLUSH: Rewrite, falls noch veränderter Datenblock im Filepuffer

```
EE94-  A2 4A       LDX   #$4A         ; $004A/4B: Pointer Datenteil des Filepuffers
EE96-  A9 10       LDA   #$10
EE98-  0D 92 FE    ORA   $FE92        ; + FCB-Index = „$D810", zeigt auf
EE9B-  A8          TAY                  Nummer „momentaner Datenblock"
EE9C-  A9 02       LDA   #$02         ; Befehl: WRITE
EE9E-  20 47 EE    JSR   $EE47        ; setzt Pointer und schreibt
EEA1-  B0 22       BCS   $EEC5
EEA3-  A9 BF       LDA   #$BF         ; Löschen von „Datablock altered" in STATUS
EEA5-  4C BC EE    JMP   $EEBC
```

### –(RE)WRITE CURRENT INDEX BLOCK

Xref: $ECCE – SET MARK: Rewrite eines veränderten Indexblocks vor neuem Read
$F556 – FLUSH: Rewrite, falls noch veränderter Indexblock im Filepuffer

```
EEA8-  20 93 EB    JSR   $EB93        ; Rewrite der VBM, falls nötig
EEAB-  A2 48       LDX   #$48         ; $0048/49: Pointer Indexteil des Filepuffers
EEAD-  A9 0E       LDA   #$0E
EEAF-  0D 92 FE    ORA   $FE92        ; + FCB-Index = „$D80E", zeigt auf
EEB2-  A8          TAY                  Nummer „momentaner Indexblock"
EEB3-  A9 02       LDA   #$02         ; Befehl: WRITE
EEB5-  20 47 EE    JSR   $EE47        ; setzt Pointer und schreibt
EEB8-  B0 0B       BCS   $EEC5
EEBA-  A9 7F       LDA   #$7F         ; Löschen von „Indexblock altered" in STATUS

EEBC-  AC 92 FE    LDY   $FE92        ; FCB-Index
EEBF-  39 08 D8    AND   $D808,Y
EEC2-  99 08 D8    STA   $D808,Y      ; STATUS
EEC5-  60          RTS
```

## $C0 – OPEN COMMAND ; Copy & Verify Pathname vorher

```
EEC6-  20 A3 E5    JSR   $E5A3        ; GET FILE
EEC9-  90 04       BCC   $EECF
EECB-  C9 40       CMP   #$40         ; INVALID PATHNAME, d.h. OPEN einer VolDIR?
EECD-  D0 07       BNE   $EED6        ; andere Fehler werden nicht abgefangen
EECF-  20 B3 EF    JSR   $EFB3        ; Sucht den FCB nach Files mit WRITE ACCESS
EED2-  90 04       BCC   $EED8          ab und kehrt zurück mit SEC, wenn schon OPEN
EED4-  A9 50       LDA   #$50         ; ERR#50: FILE (ALREADY) OPEN
EED6-  38          SEC
EED7-  60          RTS
EED8-  AC 92 FE    LDY   $FE92        ; FCB-Index von $E1A9 (freier Eintrag im FCB)
EEDB-  AD 93 FE    LDA   $FE93        ; „freier Eintrag gefunden" von $E1A9
EEDE-  D0 04       BNE   $EEE4
EEE0-  A9 42       LDA   #$42         ; ERR#42: FCB TABLE FULL
EEE2-  38          SEC
EEE3-  60          RTS
EEE4-  A2 1F       LDX   #$1F
EEE6-  A9 00       LDA   #$00
EEE8-  99 00 D8    STA   $D800,Y      ; Der FCB-Eintrag wird gelöscht. Dabei
EEEB-  C8          INY                  werden auch „Position jetzt", „Blocks
EEEC-  CA          DEX                  momentan im Speicher" etc auf Null gesetzt
EEED-  10 F9       BPL   $EEE8
EEEF-  A9 06       LDA   #$06
EEF1-  AA          TAX
EEF2-  0D 92 FE    ORA   $FE92        ; FCB-Index
EEF5-  A8          TAY                  $F801    <= $FE59:    Unitnummer
EEF6-  BD 58 FE    LDA   $FE58,X        $F802/03 <= $FE5A/5B: Key Blockno DIR
EEF9-  99 00 D8    STA   $D800,Y        $F804/05 <= $FE5C/5D: Blockno DIR-Eintrag
EEFC-  88          DEY                  $F806    <= $FE5E:    Eintragsnummer
EEFD-  CA          DEX
EEFE-  D0 F6       BNE   $EEF6
EF00-  AD 5F FE    LDA   $FE5F        ; Storage Type ($10/$20/$30/$D0) + Namenslänge
EF03-  4A          LSR                  GET FILE setzt $D0 für VolDIRs, s. dort
EF04-  4A          LSR
EF05-  4A          LSR
EF06-  4A          LSR
EF07-  AA          TAX
EF08-  99 07 D8    STA   $D807,Y      ; Storage Type im FCB ($01/$02/$03/$0D)
EF0B-  AD 7D FE    LDA   $FE7D        ; ACCESS des Files
EF0E-  29 03       AND   #$03         ; nur READ und WRITE ACCESS werden kopiert
EF10-  E0 0D       CPX   #$0D         ; Storage Type: ist File (Sub)directory?
EF12-  D0 02       BNE   $EF16
EF14-  29 01       AND   #$01         ; Ja, nur READ erlaubt
EF16-  99 09 D8    STA   $D809,Y      ; ACCESS im FCB-Eintrag
EF19-  29 02       AND   #$02         ; WRITE erlaubt?
EF1B-  F0 05       BEQ   $EF22        ; Ja, Sprung
EF1D-  AD 97 FE    LDA   $FE97        ; „File gefunden" von $EFB3 („READ ONLY")
EF20-  D0 B2       BNE   $EED4        ; Ja, Abbruch – s.„Spezielles"
EF22-  AD 7C FE    LDA   $FE7C        ; MLI VERSION des Files
                   ********************* ; wird nicht mehr geprüft! (vgl. Ver. 1.0.x)
EF25-  A9 00       LDA   #$00
EF27-  F0 04       BEQ   $EF2D        ; das ist ein „always"!
EF29-  A9 4A       LDA   #$4A         ; ERR#4A: INCOMPATIBLE FILE FORMAT
EF2B-  38          SEC
EF2C-  60          RTS
EF2D-  E0 04       CPX   #$04         ; Storage Type
EF2F-  90 04       BCC   $EF35        ; Sprung für die Typen 1/2/3
EF31-  E0 0D       CPX   #$0D         ; (Sub)directory?
EF33-  D0 F4       BNE   $EF29        ; weder noch: ERR#4A
EF35-  A2 06       LDX   #$06
```

```
EF37-   85 47       STA   $47
EF39-   AD 92 FE    LDA   $FE92
EF3C-   1D 08 FE    ORA   $FE08,X       ; ($0C/$0D/$18/$19/$15/$16/$17)
EF3F-   A8          TAY
EF40-   BD 70 FE    LDA   $FE70,X         $F80C/0D <= $FE70/71: „erster Block"
EF43-   99 00 D8    STA   $D800,Y         $F818/19 <= $FE72/73: BLOCKS USED
EF46-   CA          DEX                   $F815-17 <= $FE74-76: EOF des Files
EF47-   10 EE       BPL   $EF37
EF49-   85 46       STA   $46
EF4B-   AC 92 FE    LDY   $FE92         ; FCB-Index
EF4E-   AD 9A FE    LDA   $FE9A         ; FCB-Eintragsnummer von $EFB3 (Index/$20)
EF51-   99 00 D8    STA   $D800,Y       ; $F800: FCB-Eintragsnummer („belegt"-Flag)
EF54-   20 ED FB    JSR   $FBED         ; ALLOCATE FILEBUFFER
EF57-   B0 24       BCS   $EF7D         ; Sprung bei Kollision mit System Bit Map
EF59-   20 EB E1    JSR   $E1EB         ; Setzt die Pufferadresse und den VCB-Index
EF5C-   AD 94 BF    LDA   $BF94
EF5F-   99 1B D8    STA   $D81B,Y       ; LEVEL für FLUSH und CLOSE
EF62-   B9 07 D8    LDA   $D807,Y       ; Storage Type
EF65-   C9 04       CMP   #$04
EF67-   B0 2B       BCS   $EF94         ; Sprung für Subdirectories
EF69-   A9 FF       LDA   #$FF          ; Die „Position jetzt" wird auf FF 00 00,
EF6B-   99 14 D8    STA   $D814,Y         die „Position gewünscht" auf 00 00 00
EF6E-   A0 02       LDY   #$02            gesetzt. Der folgende Aufruf von SET MARK
EF70-   A9 00       LDA   #$00            erzwingt damit das Einlesen des ersten
EF72-   99 AA FE    STA   $FEAA,Y         Datenblocks (+ Indexblock, wenn Type > 1)
EF75-   88          DEY                   in den Filepuffer und setzt die
EF76-   10 FA       BPL   $EF72           entsprechenden Blocknr. im FCB-Eintrag
EF78-   20 48 EC    JSR   $EC48         ; SET MARK
EF7B-   90 1C       BCC   $EF99         ; ok.

EF7D-   48          PHA                 ; Fehler: Zwischenspeichern der ERR#
EF7E-   AC 92 FE    LDY   $FE92         ; FCB-Index
EF81-   B9 0B D8    LDA   $D80B,Y       ; Index der Filepuffer-Startadresse
EF84-   F0 06       BEQ   $EF8C         ; kein Puffer zugeordnet (von $E14D)
EF86-   20 4A FC    JSR   $FC4A         ; DEALLOCATE FILEBUFFER
EF89-   AC 92 FE    LDY   $FE92
EF8C-   A9 00       LDA   #$00          ; Ungültigmachen des FCB-Eintrags
EF8E-   99 00 D8    STA   $D800,Y
EF91-   68          PLA                 ; ERR#
EF92-   38          SEC
EF93-   60          RTS
```

– **OPEN FÜR (SUB)DIRECTORIES:** Lesen des Key Blocks

```
EF94-   20 04 EE    JSR   $EE04         ; liest den „ersten Block im File"
EF97-   B0 E4       BCS   $EF7D
```

– der erste Datenblock des Files ist im Filepuffer

```
EF99-   AE 91 FE    LDX   $FE91         ; VCB-Index
EF9C-   FE 1E D9    INC   $D91E,X       ; Erhöhung der Anzahl der OPEN Files
EF9F-   BD 11 D9    LDA   $D911,X       ; und Setzen des „File(s) OPEN" Flag
EFA2-   09 80       ORA   #$80
EFA4-   9D 11 D9    STA   $D911,X
EFA7-   AC 92 FE    LDY   $FE92         ; FCB-Index
EFAA-   B9 00 D8    LDA   $D800,Y
EFAD-   A0 05       LDY   #$05          ; die Refno wird in den Parameterblock
EFAF-   91 40       STA   ($40),Y         des „user"-Programms gespeichert
EFB1-   18          CLC
EFB2-   60          RTS                 ; => $DE78
```

### –SCAN FCB

Sucht den FCB nach einem bestimmten File und einem freien Eintrag ab.
Wird der File im FCB gefunden, wird ein Flag ($FE97) gesetzt.
Hat der File WRITE ACCESS, RTS mit Carry gesetzt.
Der erste freie FCB-Eintrag wird festgehalten (Nummer + Flag).

Xref: $EECF – OPEN: Ist File bereits OPEN?
$F8D0 – RENAME: nicht möglich bei offenen Files
$F96B – DESTROY: nicht möglich bei offenen Files

```
EFB3-  A9 00       LDA  #$00
EFB5-  8D 9A FE    STA  $FE9A        ; Nummer des gefundenen (freien) Eintrags
EFB8-  8D 97 FE    STA  $FE97        ; Reset des Flag: „File gefunden"
EFBB-  8D 93 FE    STA  $FE93        ; Reset des Flag: „freier Eintrag gefunden"
EFBE-  A8          TAY
EFBF-  AE 93 FE    LDX  $FE93
EFC2-  D0 03       BNE  $EFC7        ; bereits ein freier Eintrag gefunden
EFC4-  EE 9A FE    INC  $FE9A        ; ansonsten Nummer vor Test erhöhen
EFC7-  B9 00 D8    LDA  $D800,Y
EFCA-  D0 0D       BNE  $EFD9        ; Eintrag nicht frei: Sprung zum Vergleich
EFCC-  8A          TXA
EFCD-  D0 28       BNE  $EFF7        ; bereits freier Eintrag gefunden: NEXT
EFCF-  8C 92 FE    STY  $FE92        ; ansonsten wird der FCB-Index gesetzt
EFD2-  A9 FF       LDA  #$FF
EFD4-  8D 93 FE    STA  $FE93        ; und das Flag „freier Eintrag gefunden"
EFD7-  D0 1E       BNE  $EFF7        ; „always"
EFD9-  98          TYA               ; Vergleich von:
EFDA-  09 06       ORA  #$06           $FE59    – $F800:    Unitnummer
EFDC-  A8          TAY                 $FE5A/5B – $F801/02: Blockno DIR Key
EFDD-  A2 06       LDX  #$06           $FE5C/5D – $F803/04: Blockno Parent DIR
EFDF-  B9 00 D8    LDA  $D800,Y        $FE5E    – $F805:    Eintragsnummer
EFE2-  DD 58 FE    CMP  $FE58,X        $FE5F    – $F807:    Storage Type
EFE5-  D0 10       BNE  $EFF7
EFE7-  88          DEY
EFE8-  CA          DEX
EFE9-  D0 F4       BNE  $EFDF
EFEB-  EE 97 FE    INC  $FE97        ; Setzen von „File gefunden"
EFEE-  B9 09 D8    LDA  $D809,Y      ; ACCESS im FCB
EFF1-  29 02       AND  #$02
EFF3-  F0 02       BEQ  $EFF7        ; kein WRITE ACCESS: Exit ohne Carry
EFF5-  38          SEC               ; WRITE ACCESS: Abbruch mit Carry gesetzt
EFF6-  60          RTS
EFF7-  98          TYA               ; nächster FCB-Eintrag (NEXT)
EFF8-  29 E0       AND  #$E0
EFFA-  18          CLC
EFFB-  69 20       ADC  #$20
EFFD-  D0 BF       BNE  $EFBE
EFFF-  18          CLC               ; File nicht gefunden (nicht OPEN)
F000-  60          RTS
```

### $CA – READ COMMAND ; File muß OPEN sein – Set File Data vorher

```
F001-  20 0D F2    JSR  $F20D        ; Zieladresse vom „user" => $004E/4F
F004-  20 F2 F1    JSR  $F1F2        ; setzt Pufferlänge & Herunterzähler
F007-  48          PHA               ; und bringt $F809 (ACCESS) zurück
F008-  20 1F F2    JSR  $F21F        ; Kopie „Pos. jetzt"/Xfer Req/CMP EOF
F00B-  68          PLA               ; ACCESS
F00C-  29 01       AND  #$01         ; READ erlaubt?
F00E-  D0 04       BNE  $F014
F010-  A9 4E       LDA  #$4E         ; ERR#4E: FILE ACCESS ERROR
F012-  D0 24       BNE  $F038
```

```
F014-  90 25      BCC  $F03B       ; Carry von $F21F (Vergleich mit EOF)
F016-  AC 92 FE   LDY  $FE92       ; FCB-Index: „Position jetzt" + Request
F019-  B9 15 D8   LDA  $D815,Y       sind größer als EOF. Der Transferzähler
F01C-  ED AA FE   SBC  $FEAA         wird entsprechend begrenzt.
F01F-  8D DA FE   STA  $FEDA
F022-  8D AD FE   STA  $FEAD       ; Transfer-Zähler lo
F025-  B9 16 D8   LDA  $D816,Y
F028-  ED AB FE   SBC  $FEAB       ; „Position jetzt" - mittleres Byte
F02B-  8D DB FE   STA  $FEDB
F02E-  8D AE FE   STA  $FEAE       ; Transfer-Zähler hi
F031-  0D DA FE   ORA  $FEDA
F034-  D0 10      BNE  $F046       ; bei der Begrenzung sind noch Bytes übrig
F036-  A9 4C      LDA  #$4C        ; Begrenzt auf $00 Bytes Transfer:
F038-  4C F2 F0   JMP  $F0F2       ; ERR=4C: EOF ENCOUNTERED

F03B-  AD DA FE   LDA  $FEDA       ; Start des Transfer-Loop:
F03E-  0D DB FE   ORA  $FEDB         sind noch Bytes zu übertragen?
F041-  D0 03      BNE  $F046
F043-  4C F9 F0   JMP  $F0F9       ; Nein, Exit

F046-  20 82 FC   JSR  $FC82       ; Check des Zielgebiets mit der Bit Map
F049-  B0 ED      BCS  $F038       ; Error Handler
F04B-  20 18 F2   JSR  $F218       ; holt den Storage Type aus dem FCB-Eintrag
F04E-  C9 04      CMP  #$04
F050-  90 03      BCC  $F055
F052-  4C BB F1   JMP  $F1BB       ; READ für Directories
```

– Byte-per-Byte Transfer Loop

```
F055-  20 48 EC   JSR  $EC48       ; GET MARK
F058-  B0 DE      BCS  $F038       ; Error Handler
F05A-  20 10 F1   JSR  $F110       ; setzt NEWLINE (falls nötig) und Pointer
F05D-  20 3A F1   JSR  $F13A       ; kopiert vom Filepuffer ins Zielgebiet
F060-  70 E1      BVS  $F043       ; NEWLINE oder Transfer-Ende: EXIT
F062-  B0 F1      BCS  $F055       ; NEWLINE aktiv: nächster Block
F064-  AD AE FE   LDA  $FEAE       ; NEWLINE nicht aktiv: Test, ob noch
F067-  4A         LSR                mindestens ein Block ($200 Byte) übrig
F068-  F0 EB      BEQ  $F055       ; nein, Rest ebenfalls Byte-per-Byte
F06A-  8D AF FE   STA  $FEAF       ; Anzahl der übrigen Blocks
F06D-  20 06 F6   JSR  $F606       ; holt STATUS des Files aus dem FCB
F070-  29 40      AND  #$40        ; „Datablock altered" (WRITE vor READ)?
F072-  D0 E1      BNE  $F055       ; ja, SET MARK schreibt den Block zurück
F074-  8D B2 FE   STA  $FEB2       ; nein, „Mystery Flag" auf $00

F077-  A5 4E      LDA  $4E         ; die Pointer zum Datenteil des Filepuffers
F079-  85 4A      STA  $4A           werden auf den Zielbereich von READ
F07B-  A5 4F      LDA  $4F           gesetzt. Sie müssen am Ende von READ
F07D-  85 4B      STA  $4B           wieder auf den Filepuffer gesetzt werden!
```

– Block-per-Block via SET MARK (Stufe 2)

```
F07F-  20 48 EC   JSR  $EC48       ; SET MARK liest den nächsten Block
F082-  B0 69      BCS  $F0ED         direkt ins Zielgebiet von READ (Pointer!)
```

– Start des Loop für Stufe 3

```
F084-  E6 4B      INC  $4B         ; Zielpointer + $200
F086-  E6 4B      INC  $4B
F088-  CE AE FE   DEC  $FEAE       ; Transfer-Zähler - $200
F08B-  CE AE FE   DEC  $FEAE
F08E-  EE AB FE   INC  $FEAB       ; „Position gewünscht" + $200
F091-  EE AB FE   INC  $FEAB
```

```
F094-  D0 09      BNE   $F09F
F096-  EE AC FE   INC   $FEAC        ; MSByte „Position gewünscht" muß ggf.
F099-  AD AC FE   LDA   $FEAC          erhöht werden: sind wir am Ende eines
F09C-  49 01      EOR   #$01           Indexblocks? (MSByte = xxxx xx10)
F09E-  4A         LSR             ** ; wenn ja, ist das Carry jetzt gesetzt
F09F-  CE AF FE   DEC   $FEAF        ; Zähler für die übrigen Blocks
F0A2-  D0 0D      BNE   $F0B1
F0A4-  20 AD F1   JSR   $F1AD        ; keine ganzen Blocks mehr übrig, die
F0A7-  AD AD FE   LDA   $FEAD          Pointer werden zurückgesetzt, danach
F0AA-  0D AE FE   ORA   $FEAE          Test, ob noch einzelne Bytes übrig sind
F0AD-  F0 4A      BEQ   $F0F9        ; nein, Exit
F0AF-  D0 A4      BNE   $F055        ; ja, Transfer Byte-per-Byte
```

– Stufe 3: Test auf „direct access"

```
F0B1-  B0 CC      BCS   $F07F      ** ; neuer Indexblock benötigt => SET MARK
F0B3-  AD AC FE   LDA   $FEAC        ; MSByte „Position gewünscht"
F0B6-  4A         LSR
F0B7-  AD AB FE   LDA   $FEAB        ; mittleres Byte
F0BA-  6A         ROR
F0BB-  A8         TAY                ; / 2 => Index zur Blocknummer
F0BC-  B1 48      LDA   ($48),Y      ; aus dem Indexblock
F0BE-  85 46      STA   $46
F0C0-  E6 49      INC   $49
F0C2-  D1 48      CMP   ($48),Y      ; Hibyte der Blocknummer
F0C4-  D0 09      BNE   $F0CF
F0C6-  C9 00      CMP   #$00     ###
F0C8-  D0 05      BNE   $F0CF        ; Block existiert
F0CA-  8D B2 FE   STA   $FEB2        ; „sparse" Block: „Mystery Flag" auf 00
F0CD-  F0 03      BEQ   $F0D2        ; „always", Blockno = 00 00
F0CF-  B1 48      LDA   ($48),Y      ; Blocknummer hi
F0D1-  18         CLC             ### ; Indikator
F0D2-  C6 49      DEC   $49
F0D4-  B0 A9      BCS   $F07F     ### ; „sparse" Block: SET MARK löscht Zielgebiet
F0D6-  85 47      STA   $47          ; Setzen der Blocknummer hi
F0D8-  AD B2 FE   LDA   $FEB2        ; „Mystery": wurde seit einem „sparse Block"
F0DB-  F0 A2      BEQ   $F07F          nichts vom Volume gelesen => SET MARK
F0DD-  18         CLC
F0DE-  08         PHP                ; Save des INT-Status
F0DF-  78         SEI
F0E0-  A5 4B      LDA   $4B
F0E2-  85 45      STA   $45          ; Zieladresse für die RWTB
F0E4-  20 DA DE   JSR   $DEDA        ; DO DISK I/O: RWTB-Aufruf via Unitnummer
F0E7-  B0 03      BCS   $F0EC
F0E9-  28         PLP
F0EA-  90 98      BCC   $F084        ; "always": Loop

F0EC-  28         PLP                ; Reload des INT-Status
F0ED-  48         PHA                ; ERR#
F0EE-  20 AD F1   JSR   $F1AD        ; Zurücksetzen der Pointer auf Filepuffer
F0F1-  68         PLA                ; ERR#
```

– Error Handler

Xref: innerhalb von READ: mehrere Sprünge hierher
$F2DE – WRITE Error Exit

```
F0F2-  48         PHA                ; ERR#
F0F3-  20 F9 F0   JSR   $F0F9        ; Xfer Count => „user"-Parameter, SET MARK
F0F6-  68         PLA
F0F7-  38         SEC
F0F8-  60         RTS
```

– Setzt den „Transfer Count" und bringt den Filepuffer auf den neuen Stand. Nach READ-Stufen 2 und 3 ist die „Position jetzt" im FCB mindestens einen ganzen Block von der „Position gewünscht" entfernt stehengeblieben, d.h. der nächste Aufruf von SET MARK lädt den Filepuffer wieder mit den der neuen Position im File entsprechenden Daten.

Xref: $F043 – JP READ Exit, Transfer beendet
$F0F3 – READ: Error Handler
$F1C8 – READ für Directories: JSR nach Transfer-Ende, danach Test auf EOF
JP $F2A6 – WRITE Exit: Request Count = 00 00
JP $F326 – WRITE Exit: Falls auf Block-Ende => SET MARK für Rewrite des Blocks

```
F0F9-   A0 06      LDY   #$06          ; Bytes $06/$07 im „user"-Parameterblock
F0FB-   38         SEC                   für den Transfer Count
F0FC-   AD DA FE   LDA   $FEDA         ; Transfer Request
F0FF-   ED AD FE   SBC   $FEAD         ; minus Herunterzähler = Anzahl der
F102-   91 40      STA   ($40),Y         transferierten Bytes
F104-   C8         INY
F105-   AD DB FE   LDA   $FEDB
F108-   ED AE FE   SBC   $FEAE         ; Hibytes
F10B-   91 40      STA   ($40),Y
F10D-   4C 48 EC   JMP   $EC48         ; SET MARK – RTS von da aus ($DE78/intern)
```

– Justiert den Quell/Zielpointer „rückwärts", damit für den Transfer-Loop derselbe Index benutzt werden kann. Kopie von NEWLINE, wenn spezifiziert.

Xref: $F05A – READ: vor der Kopierroutine Stufe 1 ($E330)
$F1C0 – READ für Directories – vor der Kopierroutine ($E330)
$F31E – WRITE: vor der WRITE-Kopierroutine ($E51F)

```
F110-   AC 92 FE   LDY   $FE92         ; FCB-Index
F113-   38         SEC
F114-   A5 4E      LDA   $4E           ; Quell/Zielpointer lo
F116-   ED AA FE   SBC   $FEAA         ; „Position jetzt" lo
F119-   85 4E      STA   $4E
F11B-   B0 02      BCS   $F11F
F11D-   C6 4F      DEC   $4F           ; Quell/Zielpointer hi
F11F-   B9 1F D8   LDA   $D81F,Y       ; NEWLINE AND MASK
F122-   18         CLC
F123-   F0 0A      BEQ   $F12F         ; nicht spezifiziert
F125-   38         SEC                 ; Indikator: NEWLINE aktiv
F126-   8D B1 FE   STA   $FEB1
F129-   B9 0A D8   LDA   $D80A,Y       ; NEWLINE CHAR
F12C-   8D B0 FE   STA   $FEB0
F12F-   AC AA FE   LDY   $FEAA         ; Start-Index für Kopierroutine
F132-   A5 4A      LDA   $4A           ; Kopie des Ziel/Quellpointers, wird
F134-   85 4C      STA   $4C             beim Loop verändert
F136-   AE AD FE   LDX   $FEAD         ; Transferzähler lo
F139-   60         RTS                 ; mit SEC, wenn NEWLINE aktiv ist
```

**– READ MOVER LOOP**

Aufruf mit SEC, wenn NEWLINE aktiv. Exit mit V-Flag gesetzt, wenn Abbruch via NEWLINE oder Transfer Request heruntergezählt. Das Carry bleibt immer unverändert!

Xref: $F05D – READ Byte-per-Byte für Datenfiles
$F1C3 – READ für Directories

```
F13A-  8A          TXA                ; Aufruf mit Transferzähler lo im X-Reg
F13B-  D0 08       BNE   $F145
F13D-  AD AE FE    LDA   $FEAE        ; Transferzähler hi
F140-  F0 52       BEQ   $F194        ; auf $00 heruntergezählt: Exit
F142-  CE AE FE    DEC   $FEAE
F145-  CA          DEX
F146-  B1 4C       LDA   ($4C),Y      ; $004C/4D: Quell-Pointer (Filepuffer)
F148-  91 4E       STA   ($4E),Y      ; $004E/4F: Pointer ins Zielgebiet
F14A-  8A          TXA
F14B-  F0 1B       BEQ   $F168        ; wenn letztes Byte im Transferzähler (lo)
F14D-  B0 2E       BCS   $F17D        ; wenn NEWLINE aktiv, Test
F14F-  CA          DEX
F150-  C8          INY
F151-  D0 F3       BNE   $F146        ; sonst weiter, bis Y-Index = $00

F153-  A5 4D       LDA   $4D
F155-  E6 4F       INC   $4F          ; Zieladresse + $100
F157-  EE AB FE    INC   $FEAB        ; „Position jetzt" + $100
F15A-  D0 03       BNE   $F15F
F15C-  EE AC FE    INC   $FEAC        ; MSByte „Position jetzt"
F15F-  E6 4D       INC   $4D          ; Quell-Pointer
F161-  45 4B       EOR   $4B          ; $004D EOR $004B => $00, wenn erste Seite
F163-  F0 E1       BEQ   $F146          innerhalb des Transfers => 2. Seite
F165-  B8          CLV                ; Indikator: nicht durch NL/Zähler beendet
F166-  50 2F       BVC   $F197        ; „always" – Filepuffer-Ende

F168-  AD AE FE    LDA   $FEAE        ; Herunterzähler low = $00. Test Zähler hi
F16B-  F0 1A       BEQ   $F187        ; wenn auch $00, Exit mit V-Flag gesetzt
F16D-  C8          INY                ; noch Bytes übrig:
F16E-  D0 06       BNE   $F176        ; wenn nicht Seitenende, NEXT
F170-  A5 4D       LDA   $4D
F172-  45 4B       EOR   $4B          ; erste oder zweite Seite des Transfers?
F174-  D0 03       BNE   $F179        ; Sprung wenn zweite Seite
F176-  CE AE FE    DEC   $FEAE        ; erste Seite: Transferzähler hi minus 1
F179-  88          DEY                ; „INY" rückgängig machen
F17A-  4C 4D F1    JMP   $F14D        ; nächste Seite

F17D-  B1 4C       LDA   ($4C),Y      ; gerade kopiertes Byte
F17F-  2D B1 FE    AND   $FEB1        ; NEWLINE AND MASK
F182-  4D B0 FE    EOR   $FEB0        ; NEWLINE CHAR
F185-  D0 C8       BNE   $F14F        ; kein NEWLINE: NEXT
F187-  C8          INY
F188-  D0 0A       BNE   $F194        ; $004E/4F auf nächstes Byte setzen
F18A-  E6 4F       INC   $4F
F18C-  EE AB FE    INC   $FEAB        ; und „Position gewünscht" anpassen, falls
F18F-  D0 03       BNE   $F194          eine Seitengrenze überschritten wurde
F191-  EE AC FE    INC   $FEAC

F194-  2C AC F1    BIT   $F1AC        ; = $60, SET V-FLAG (Transfer-Ende)

F197-  8C AA FE    STY   $FEAA        ; „Position gewünscht"- LoByte
F19A-  70 01       BVS   $F19D        ; wenn Exit durch Filepuffer-Ende (V gelöscht)
F19C-  E8          INX                  ist X um eins zu klein
F19D-  8E AD FE    STX   $FEAD        ; zwischenspeichern
F1A0-  08          PHP
F1A1-  18          CLC
F1A2-  98          TYA
F1A3-  65 4E       ADC   $4E
F1A5-  85 4E       STA   $4E          ; $004E/4F auf Position nach Ende
F1A7-  90 02       BCC   $F1AB          der Kopieraktion
F1A9-  E6 4F       INC   $4F
```

```
F1AB-   28          PLP             ; mit V bei Abbruch via NEWLINE oder Zähler
F1AC-   60          RTS             ; mit C falls NEWLINE aktiv
```

– Restore der Filepuffer-Pointer
Bei READ Stufe 2 und 3 zeigen die Filepuffer-Pointer auf das Zielgebiet selber und müssen nach Ende von READ wieder auf den Filepuffer gesetzt werden.

Xref: $F0A4 – READ: keine weiteren Block-per-Block Transfers
$F0EE – READ: Error in Stufe 2 oder 3

```
F1AD-   A5 4A       LDA   $4A       ; momentane Ziel-Pointer => $004E/4F, falls
F1AF-   85 4E       STA   $4E         noch Byte-per-Byte Transfers zu machen
F1B1-   A5 4B       LDA   $4B         sind
F1B3-   85 4F       STA   $4F
F1B5-   AC 92 FE    LDY   $FE92     ; FCB-Index
F1B8-   4C EB E1    JMP   $E1EB     ; Teil von Set File Data, setzt Pointer neu
```

– READ für nicht-File Typen
Der File muß „automatisch" den Typ (Sub)directory haben – sonst Error bei OPEN

Xref: JP $F052 – READ: Storage Type ist nicht im Bereich 0/1/2/3

```
F1BB-   20 48 EC    JSR   $EC48     ; SET MARK - Block lesen
F1BE-   B0 2F       BCS   $F1EF     ; READ Error Handler
F1C0-   20 10 F1    JSR   $F110     ; Pointer justieren / NEWLINE kopieren
F1C3-   20 3A F1    JSR   $F13A     ; READ MOVER LOOP
F1C6-   50 F3       BVC   $F1BB     ; Loop bis NEWLINE oder Count = $00
F1C8-   20 F9 F0    JSR   $F0F9     ; Return des Transfer Count + SET MARK
F1CB-   90 20       BCC   $F1ED     ; RTS
F1CD-   C9 4C       CMP   #$4C      ; Fehler: EOF ENCOUNTERED?
F1CF-   38          SEC
F1D0-   D0 1C       BNE   $F1EE     ; nein, anderer Fehler
```

– READ Directories, ab hier: s.„Spezielles"!

```
F1D2-   20 89 ED    JSR   $ED89     ; die Directory wurde vollständig gelesen,
F1D5-   20 67 ED    JSR   $ED67       d.h. bis zum EOF. „Position jetzt" wird
F1D8-   A0 00       LDY   #$00        auf den EOF gesetzt ($DF7C), der Datenteil
F1DA-   AE 92 FE    LDX   $FE92       des Filepuffers wird gelöscht ($DF5A).
F1DD-   BD 10 D8    LDA   $D810,X   ; Bytes $10/11 des FCB/Eintrags
F1E0-   91 4A       STA   ($4A),Y     („Block jetzt im Speicher") werden
F1E2-   A9 00       LDA   #$00        als Bytes $00/01 im Filepuffer gesetzt.
F1E4-   9D 10 D8    STA   $D810,X   ; und im FCB auf 00 00
F1E7-   E8          INX
F1E8-   C8          INY
F1E9-   C0 02       CPY   #$02
F1EB-   D0 F0       BNE   $F1DD     ; Loop
F1ED-   18          CLC
F1EE-   60          RTS
F1EF-   4C F2 F0    JMP   $F0F2     ; Error Handler
```

– Setzt die Zielpuffer-Länge aus dem Request Count und bringt ACCESS zurück

Xref: $F004 – READ Entry
$E47C – WRITE Entry

```
F1F2-   A0 04       LDY   #$04          ; Bytes $04 und $05: Request Count
F1F4-   B1 40       LDA   ($40),Y         aus dem „user"-Parameterblock
F1F6-   8D DA FE    STA   $FEDA         ; Pufferlänge
F1F9-   8D AD FE    STA   $FEAD         ; Herunterzähler
F1FC-   C8          INY
F1FD-   B1 40       LDA   ($40),Y
F1FF-   8D DB FE    STA   $FEDB         ; Hibytes
F202-   8D AE FE    STA   $FEAE
F205-   AC 92 FE    LDY   $FE92         ; FCB-Index
F208-   B9 09 D8    LDA   $D809,Y       ; ACCESS
F20B-   18          CLC
F20C-   60          RTS
```

– Kopiert die Startadresse des „user"-Puffers auf $004E/4F und bringt Storage Type zurück.

Xref: $E235 – ONLINE Entry: Puffer für Volume-Namen (Storage Type wird ignoriert)
$F001 – READ Entry: Start des Zielgebiets
$F2A9 – WRITE: Startadresse der Daten, die geschrieben werden sollen

```
F20D-   A0 02       LDY   #$02          ; Bytes $02/03 aus dem Parameterblock
F20F-   B1 40       LDA   ($40),Y         des „user"-Programms: Quell/Zielstart
F211-   85 4E       STA   $4E
F213-   C8          INY
F214-   B1 40       LDA   ($40),Y
F216-   85 4F       STA   $4F
```

**– GET STORAGE TYPE** (aus dem FCB-Eintrag)

Xref: $F04B – READ: Datenfile oder Directory?
$F39E – WRITE: Type-Check nach erster Typerweiterung
$F3E4 – WRITE: Unterscheidung im Erweiterungsprozeß '1 -> 2' und '2 -> 3'
$F426 – WRITE: Erhöhung des Storage Type im Erweiterungsprozeß
$F611 – SET EOF: Unterscheidung Datenfile – Directory

```
F218-   AC 92 FE    LDY   $FE92         ; FCB-Index
F21B-   B9 07 D8    LDA   $D807,Y       ; Byte $07 im FCB-Eintrag: Storage Type
F21E-   60          RTS
```

– Kopiert „Position jetzt" auf $FEAA/AB/AC und testet „Position + Request" gegen EOF

Xref: $F008 – READ: Test, ob Begrenzung des Request auf den EOF notwendig
$F28A – WRITE: Test, ob der EOF des Files verändert (vergrößert) werden muß

```
F21F-   A2 00       LDX   #$00
F221-   AC 92 FE    LDY   $FE92         ; FCB-Index
F224-   18          CLC
F225-   B9 12 D8    LDA   $D812,Y       ; „Position jetzt" aus dem FCB-Eintrag
F228-   9D AA FE    STA   $FEAA,X
F22B-   9D 8D FE    STA   $FE8D,X
```

```
F22E-   7D DA FE    ADC   $FEDA,X       ; Request Count - $FEDC muß $00 sein
F231-   9D 86 FE    STA   $FE86,X
F234-   8A          TXA
F235-   49 02       EOR   #$02          ; „CMP #02" unter Beibehaltung des Carry
F237-   F0 04       BEQ   $F23D
F239-   C8          INY
F23A-   E8          INX
F23B-   D0 E8       BNE   $F225
```

Xref: $F265 – WRITE Error Exit: Setzen des neuen EOF

```
F23D-   BD 86 FE    LDA   $FE86,X       ; Vergleich der letzten „Position jetzt"
F240-   D9 15 D8    CMP   $D815,Y         von SET MARK mit dem EOF (FCB-Eintrag)
F243-   90 06       BCC   $F24B         ; RTS mit wenn der EOF
F245-   D0 04       BNE   $F24B               immer noch größer ist
F247-   88          DEY
F248-   CA          DEX
F249-   10 F2       BPL   $F23D
F24B-   60          RTS                 ; wenn EOF kleiner, RTS mit Carry gesetzt
```

– Teil des WRITE Error Exit. Update von „Position jetzt" und EOF im FCB

Xref: $F2DA – WRITE Error bei Erweiterung auf Type 3

```
F24C-   20 7E F2    JSR   $F27E         ; X-Reg = FCB-Index + 02
F24F-   BD 8A FE    LDA   $FE8A,X
F252-   99 15 D8    STA   $D815,Y       ; Setzen des neuen EOF im FCB
F255-   BD 8D FE    LDA   $FE8D,X       ; Setzen der letzten „Position jetzt" von
F258-   99 12 D8    STA   $D812,Y         SET MARK als „Position jetzt" im FCB
F25B-   9D 86 FE    STA   $FE86,X         und für den Vergleich EOF/Position ($F23D)
F25E-   88          DEY
F25F-   CA          DEX
F260-   10 ED       BPL   $F24F
F262-   20 7E F2    JSR   $F27E         ; X-Reg = FCB-Index + 02
F265-   20 3D F2    JSR   $F23D     ### ; Vergleich EOF/letzte „Position jetzt"
```

Xref: $F265 – WRITE Error Exit: wenn „Position jetzt" > EOF => neuer EOF
$F28D – WRITE: Neusetzen des EOF, wenn „Position jetzt" + Request > EOF

```
F268-   20 7E F2    JSR   $F27E         ; X-Reg = FCB-Index + 02
F26B-   B9 15 D8    LDA   $D815,Y       ; der alte EOF aus dem FCB wird kopiert
F26E-   9D 8A FE    STA   $FE8A,X
F271-   90 06       BCC   $F279     ### ; von $F23D oder $F28D gesetzt!
F273-   BD 86 FE    LDA   $FE86,X       ; EOF wurde überschritten und wird mit
F276-   99 15 D8    STA   $D815,Y         letzter „Position jetzt" bzw. „Position
F279-   88          DEY                   jetzt" + Request Count neu gesetzt
F27A-   CA          DEX
F27B-   10 EE       BPL   $F26B
F27D-   60          RTS
```

– Setzt X-Reg auf FCB-Index + 2

Xref: $F24C/$F262/$F268 – s.o.

```
F27E-   A9 02       LDA   #$02
F280-   AA          TAX
F281-   0D 92 FE    ORA   $FE92         ; FCB-Index
```

```
F284-   A8          TAY
F285-   60          RTS
```

## $CB – WRITE COMMAND

```
F286-   20 F2 F1    JSR   $F1F2        ; Request Count => Pufferlänge ($FEAD/AE)
F289-   48          PHA                ; zurückgebrachter ACCESS des Files
F28A-   20 1F F2    JSR   $F21F        ; kopiert „Position jetzt" & EOF, Vergleich
F28D-   20 68 F2    JSR   $F268        ; wenn EOF überschritten: Neusetzen
F290-   68          PLA                ; ACCESS
F291-   29 02       AND   #$02         ; WRITE erlaubt (Bit 1 gesetzt)?
F293-   D0 04       BNE   $F299
F295-   A9 4E       LDA   #$4E         ; ERR#4E: FILE ACCESS ERROR
F297-   D0 40       BNE   $F2D9        ; „always"

F299-   20 58 F4    JSR   $F458        ; „ACTIVATE" - Test auf WRITE PROTECTED
F29C-   B0 3B       BCS   $F2D9              des Volumes (Laufwerk)
F29E-   AD DA FE    LDA   $FEDA        ; Request Count
F2A1-   0D DB FE    ORA   $FEDB
F2A4-   D0 03       BNE   $F2A9
F2A6-   4C F9 F0    JMP   $F0F9        ; Request Count = 00 00, Exit ohne Fehler
F2A9-   20 0D F2    JSR   $F20D        ; kopiert die Startadresse der Daten und
F2AC-   C9 04       CMP   #$04         ; bringt den Storage Type des Files zurück
F2AE-   B0 E5       BCS   $F295        ; nicht 0/1/2/3: FILE ACCESS ERROR
```

– Transfer Loop

```
F2B0-   20 48 EC    JSR   $EC48        ; SET MARK
F2B3-   B0 24       BCS   $F2D9
F2B5-   20 06 F6    JSR   $F606        ; GET STATUS
F2B8-   29 07       AND   #$07         ; Erweiterungsbits sind Bit 0/1/2
F2BA-   F0 62       BEQ   $F31E        ; keine Erweiterung notwendig
F2BC-   A0 00       LDY   #$00
F2BE-   C8          INY                ; Zählung der gesetzten Erweiterungsbits
F2BF-   4A          LSR                  in STATUS (max. 3)
F2C0-   D0 FC       BNE   $F2BE
F2C2-   8C 94 FE    STY   $FE94        ; = benötigte Blocks (BLOCKS NEEDED) lo
F2C5-   8D 95 FE    STA   $FE95        ; = BLOCKS NEEDED (Hibyte, immer $00)
F2C8-   20 73 E9    JSR   $E973        ; SPACE ON VOLUME? (Test - keine Belegung)
F2CB-   B0 0C       BCS   $F2D9        ; Sprung, wenn nicht mehr genug Platz
F2CD-   20 06 F6    JSR   $F606        ; GET STATUS
F2D0-   29 04       AND   #$04         ; Erweiterung auf Type 3?
F2D2-   F0 0D       BEQ   $F2E1        ; nein
F2D4-   20 99 F3    JSR   $F399        ; erweitert das File auf Type 3
F2D7-   90 14       BCC   $F2ED
F2D9-   48          PHA                ; Error Exit: Save der ERR#
F2DA-   20 4C F2    JSR   $F24C        ; Error Handler: setzt EOF und FCB-Eintrag
F2DD-   68          PLA
F2DE-   4C F2 F0    JMP   $F0F2        ; Error Handler von READ
```

– Test, ob neuer Indexblock benötigt (Type 3 und Erweiterung Type 1 => 2)

```
F2E1-   20 06 F6    JSR   $F606        ; GET STATUS
F2E4-   29 02       AND   #$02         ; (irgendein) Indexblock benötigt?
F2E6-   F0 05       BEQ   $F2ED        ; nein
F2E8-   20 E4 F3    JSR   $F3E4        ; erstellt einen Indexblock
F2EB-   B0 EC       BCS   $F2D9          und erhöht ggf. den Storage Type
```

– Erweiterungen des Storage Type erledigt – File hat jetzt mindestens Type 2

```
F2ED-   20 38 F4    JSR   $F438        ; belegt den nächsten freien Block
F2F0-   B0 E7       BCS   $F2D9              (Blocknummer auf $FE86/87)
F2F2-   20 06 F6    JSR   $F606        ; GET STATUS
F2F5-   09 80       ORA   #$80         ; Setzen von „Indexblock altered"
F2F7-   29 F8       AND   #$F8         ; und Löschen der Erweiterungsbits
F2F9-   99 08 D8    STA   $D808,Y
F2FC-   AD AC FE    LDA   $FEAC        ; „Position jetzt" – MSByte
F2FF-   4A          LSR
F300-   AD AB FE    LDA   $FEAB        ; „Position jetzt" – mittleres Byte
F303-   6A          ROR
F304-   A8          TAY                ; /2 = Block-Position im Indexblock
F305-   E6 49       INC   $49
F307-   AD 87 FE    LDA   $FE87        ; von ALLOCATE NEXT BLOCK auf die Nummer
F30A-   AA          TAX                  des nächsten freien Blocks gesetzt:
F30B-   91 48       STA   ($48),Y        die Blocknummer wird im Indexblock
F30D-   C6 49       DEC   $49            eingetragen. Der Datenteil des Filepuffers
F30F-   AD 86 FE    LDA   $FE86          ist bereits von SET MARK gelöscht worden
F312-   91 48       STA   ($48),Y
F314-   AC 92 FE    LDY   $FE92        ; FCB-Index
F317-   99 10 D8    STA   $D810,Y      ; Nummer „Datenblock momentan im Speicher"
F31A-   8A          TXA
F31B-   99 11 D8    STA   $D811,Y      ; Hibyte
```

– der zu beschreibende Datenblock ist im Speicher

```
F31E-   20 10 F1    JSR   $F110        ; Quell-Pointer justieren (s. READ)
F321-   20 29 F3    JSR   $F329        ; WRITE MOVER LOOP – NEWLINE wird ignoriert
F324-   50 8A       BVC   $F2B0        ; noch Bytes übrig => nächstes SET MARK
F326-   4C F9 F0    JMP   $F0F9        ; gemeinsames Ende von READ und WRITE
```

**– WRITE MOVER LOOP**

Xref: $F321 – s.o.

```
F329-   8A          TXA                ; Aufruf mit Transferzähler low in X-Reg
F32A-   D0 08       BNE   $F334
F32C-   AD AE FE    LDA   $FEAE        ; Transferzähler hi
F32F-   F0 45       BEQ   $F376        ; Exit
F331-   CE AE FE    DEC   $FEAE        ; Transferzähler minus $100
F334-   CA          DEX
F335-   B1 4E       LDA   ($4E),Y      ; Quelle – justiert auf Seitengrenze
F337-   91 4C       STA   ($4C),Y      ; Datenteil des Filepuffers
F339-   8A          TXA
F33A-   F0 18       BEQ   $F354
F33C-   C8          INY
F33D-   D0 F5       BNE   $F334        ; Loop bis zur nächsten Seitengrenze

F33F-   A5 4D       LDA   $4D
F341-   E6 4F       INC   $4F          ; Quelle + $100
F343-   EE AB FE    INC   $FEAB        ; nächste „Position gewünscht" + $100
F346-   D0 03       BNE   $F34B
F348-   EE AC FE    INC   $FEAC        ; MSByte „Position gewünscht"
F34B-   E6 4D       INC   $4D
F34D-   45 4B       EOR   $4B          ; war das die erste Speicherseite?
F34F-   F0 E3       BEQ   $F334        ; ja, nächste Seite
F351-   B8          CLV                ; nein, Indikator: noch Bytes übrig
F352-   50 25       BVC   $F379        ; „always" – nächster Block (SET MARK)

F354-   AD AE FE    LDA   $FEAE        ; Transferzähler lo ist $00. Wenn der
F357-   F0 10       BEQ   $F369          Zähler hi auch $00 ist => Exit
F359-   C8          INY                ; nein, Index wird erhöht
```

```
F35A-   D0 06       BNE   $F362        ; nächste Seite
F35C-   A5 4D       LDA   $4D          ; $100 oder $200 Bytes transferiert:
F35E-   45 4B       EOR   $4B            wenn das die zweiten $100 Bytes waren,
F360-   D0 03       BNE   $F365          ist der Filepuffer voll => Sprung
F362-   CE AE FE    DEC   $FEAE        ; ansonsten Transferzähler minus $100
F365-   88          DEY
F366-   4C 3C F3    JMP   $F33C        ; Loop bis Filepuffer voll/Zähler-Ende

F369-   C8          INY                ; Sprung hierher, wenn Zähler = 00 00
F36A-   D0 0A       BNE   $F376        ; wenn das auf einer Seitengrenze passiert,
F36C-   E6 4F       INC   $4F            wird der Quell-Pointer und die
F36E-   EE AB FE    INC   $FEAB          nächste „Position gewünscht" für SET MARK
F371-   D0 03       BNE   $F376          (WRITE Exit) ein letztes Mal erhöht
F373-   EE AC FE    INC   $FEAC
F376-   2C AC F1    BIT   $F1AC        ; $60 => SET V-FLAG (Transfer-Ende)
F379-   8C AA FE    STY   $FEAA
F37C-   8E AD FE    STX   $FEAD
F37F-   08          PHP                ; „Push V-Flag"
F380-   20 06 F6    JSR   $F606        ; GET STATUS
F383-   09 50       ORA   #$50         ; Bit 6: „Datablock altered" - Bit 4: Flag für
F385-   99 08 D8    STA   $D808,Y        „ACTIVATE" und FLUSH (File verändert)
F388-   18          CLC
F389-   AD AA FE    LDA   $FEAA
F38C-   65 4E       ADC   $4E          ; Justieren des Quell-Pointers auf
F38E-   85 4E       STA   $4E            Transfer-Ende + 1
F390-   90 02       BCC   $F394
F392-   E6 4F       INC   $4F
F394-   20 66 FA    JSR   $FA66        ; setzt Byte $1C: „File altered"
F397-   28          PLP                ; „Pull V-Flag"
F398-   60          RTS
```

– Erweiterung auf Storage Type 3

Xref: $F2D4 – WRITE: Erweiterungsbit 2 in STATUS durch SET MARK gesetzt

```
F399-   20 F1 F3    JSR   $F3F1        ; Erhöhung des Storage Type um 1
F39C-   B0 52       BCS   $F3F0
F39E-   20 18 F2    JSR   $F218        ; holt Storage Type aus dem FCB
F3A1-   C9 03       CMP   #$03         ; jetzt Type 3?
F3A3-   F0 05       BEQ   $F3AA
F3A5-   20 F1 F3    JSR   $F3F1        ; nein, noch eine Erhöhung
F3A8-   B0 46       BCS   $F3F0
```

– Erstellung eines neuen/nächsten Indexblocks (und/oder Erweiterung Type 2 => 3)

```
F3AA-   20 38 F4    JSR   $F438        ; belegt den nächsten freien Block
F3AD-   B0 41       BCS   $F3F0          für den neuen Indexblock
F3AF-   AD AC FE    LDA   $FEAC        ; „Position gewünscht" - MSByte
F3B2-   4A          LSR
F3B3-   A8          TAY                ; => Indexblock-Position im Masterblock
F3B4-   AD 86 FE    LDA   $FE86        ; Nummer des nächsten freien Blocks ($E62E)
F3B7-   AA          TAX
F3B8-   91 48       STA   ($48),Y      ; die Blocknummer wird im Indexteil des
F3BA-   E6 49       INC   $49            des Filepuffers eingetragen. Hatte der
F3BC-   AD 87 FE    LDA   $FE87          File vorher den Type 2, wurde dieser Teil
F3BF-   91 48       STA   ($48),Y        von SET MARK gelöscht und enthält jetzt
F3C1-   C6 49       DEC   $49            einen einzigen verzeichneten Indexblock.
F3C3-   AC 92 FE    LDY   $FE92        ; FCB-Index
F3C6-   99 0F D8    STA   $D80F,Y      ; Nummer „Indexblock momentan im Speicher"
F3C9-   8A          TXA
F3CA-   99 0E D8    STA   $D80E,Y      ; Blocknummer hi
```

```
F3CD-  20 8E EE   JSR  $EE8E       ; Rewrite des VBM-Blocks und des Indexteils
F3D0-  B0 1E      BCS  $F3F0         des Filepuffers (MASTER-Block)
F3D2-  A0 00      LDY  #$00
F3D4-  98         TYA
F3D5-  91 48      STA  ($48),Y     ; Löschen des Indexteils im Filepuffer:
F3D7-  C8         INY                hier kommt der neue Indexblock hinein
F3D8-  D0 FB      BNE  $F3D5
F3DA-  E6 49      INC  $49
F3DC-  91 48      STA  ($48),Y
F3DE-  C8         INY                der Indexblock ist damit belegt und
F3DF-  D0 FB      BNE  $F3DC         momentan noch leer
F3E1-  C6 49      DEC  $49
F3E3-  60         RTS
```

– wenn Type 1, Erweiterung auf Type 2, ansonsten Lesen des „ersten Blocks im File", d.h. des MASTER- oder Indexblocks

Xref: $F2E8 – WRITE: SET MARK hat in STATUS die Erweiterungsbits 0/1 gesetzt.

```
F3E4-  20 18 F2   JSR  $F218       ; holt den momentanen Storage Type
F3E7-  C9 01      CMP  #$01
F3E9-  F0 06      BEQ  $F3F1       ; Sprung für Type 1
F3EB-  20 3B EE   JSR  $EE3B       ; sonst Lesen des „ersten Blocks im File"
F3EE-  90 BA      BCC  $F3AA
F3F0-  60         RTS
```

– Erhöhung des Storage Type um 1 – Typerweiterung

Xref: $F3E9 – Type 1 => Type 2
$F3EE – Type 2 => Type 3 oder neuer Indexblock für Type 3
$F399/F3A5 – Erweiterung auf Type 3

```
F3F1-  20 38 F4   JSR  $F438       ; belegt den nächsten freien Block
F3F4-  B0 41      BCS  $F437         (Nummer auf $FE86/87)
F3F6-  AC 92 FE   LDY  $FE92       ; FCB-Index
F3F9-  B9 0C D8   LDA  $D80C,Y     ; und setzt die Nummer des gerade belegten
F3FC-  48         PHA                Blocks als „erster Block im File" und als
F3FD-  AD 86 FE   LDA  $FE86         Nummer „Indexblock momentan im Speicher"
F400-  AA         TAX                Die vorherige Nummer „erster Block"
F401-  99 0C D8   STA  $D80C,Y       wird zwischengespeichert.
F404-  B9 0D D8   LDA  $D80D,Y
F407-  48         PHA
F408-  AD 87 FE   LDA  $FE87
F40B-  99 0D D8   STA  $D80D,Y     ; „erster Block im File" – Hibyte
F40E-  99 0F D8   STA  $D80F,Y     ; „Indexblock momentan im Speicher" – Hibyte
F411-  8A         TXA
F412-  99 0E D8   STA  $D80E,Y
F415-  A0 00      LDY  #$00
F417-  E6 49      INC  $49         ; der Indexteil des Filepuffers wurde von
F419-  68         PLA                SET MARK gelöscht. Die vorherige Nummer
F41A-  91 48      STA  ($48),Y       des „ersten Blocks im File" landet
F41C-  C6 49      DEC  $49           als erster Eintrag im Indexteil.
F41E-  68         PLA
F41F-  91 48      STA  ($48),Y
F421-  20 8E EE   JSR  $EE8E       ; Rewrite des Indexteils im Filepuffer
F424-  B0 11      BCS  $F437
F426-  20 18 F2   JSR  $F218       ; holt den Storage Type des Files
F429-  69 01      ADC  #$01
F42B-  99 07 D8   STA  $D807,Y     ; STORAGE TYPE WIRD ERHÖHT
```

```
F42E-   B9 08 D8    LDA   $D808,Y      ; STATUS
F431-   09 08       ORA   #$08         ; Setzen von Bit 3, wird nirgendwo
F433-   99 08 D8    STA   $D808,Y        wieder ausgewertet (s. „Spezielles")
F436-   18          CLC
F437-   60          RTS
```

– Belegung des nächsten freien Blocks für ein File

Xref: $F2ED – WRITE: Erweiterung erledigt, Datenblock benötigt
$F3AA – WRITE: Erweiterung: neuer Indexblock
$F3F1 – WRITE: Erhöhung des Storage Type um 1, Master oder Indexblock benötigt

```
F438-   20 B6 EA    JSR   $EAB6         ; ALLOCATE NEXT BLOCK
F43B-   B0 1A       BCS   $F457           (Blocknummer auf $FE86/87)
F43D-   20 06 F6    JSR   $F606         ; GET STATUS
F440-   09 10       ORA   #$10          ; Setzen von Bit 4 – „File altered" –
F442-   99 08 D8    STA   $D808,Y         wird von „ACTIVATE" und FLUSH benutzt
F445-   B9 18 D8    LDA   $D818,Y
F448-   18          CLC                 ; unnötig
F449-   69 01       ADC   #$01          ; Erhöhung von BLOCKS USED um eins
F44B-   99 18 D8    STA   $D818,Y
F44E-   B9 19 D8    LDA   $D819,Y
F451-   69 00       ADC   #$00
F453-   99 19 D8    STA   $D819,Y
F456-   18          CLC
F457-   60          RTS
```

**–„ACTIVATE":**
Wenn noch nicht geschehen, Test des Ziel-Volumes auf WRITE PROTECTED und „READY"

Xref: $F299 – WRITE
$F626 – SET EOF

```
F458-   20 06 F6    JSR   $F606         ; GET STATUS
F45B-   29 F0       AND   #$F0          ; wurde für diesen File schon einmal
F45D-   D0 F7       BNE   $F456           seit OPEN auf das Volume geschrieben?
F45F-   B9 01 D8    LDA   $D801,Y       ; wenn nicht, wird die Unitnummer gesetzt
F462-   8D 30 BF    STA   $BF30
```

Xref: $E31E – CREATE: Test WPROT und „READY" vor Anhängen eines Subdir-Blocks
$F998 – DESTROY: Test WPROT und „READY" vorher

```
F465-   85 43       STA   $43           ; Einsprung/Aufruf mit Unitnummer im Akku
F467-   A5 47       LDA   $47
F469-   48          PHA
F46A-   A5 46       LDA   $46           ; „alte Blocknummer" wird
F46C-   48          PHA                   zwischengespeichert
F46D-   A9 00       LDA   #$00
F46F-   85 42       STA   $42           ; „$00"-COMMAND
F471-   85 46       STA   $46           ; Blocknummer: 00 00, damit auf keinen Fall
F473-   85 47       STA   $47             die maximale Blockno überschritten wird
F475-   20 DA DE    JSR   $DEDA         ; DO DISK I/O
F478-   AA          TAX                 ; eventuelle ERR#
```

```
F479-   68          PLA
F47A-   85 46       STA   $46
F47C-   68          PLA                 ; Restore der „alten" Blocknummer
F47D-   85 47       STA   $47
F47F-   8A          TXA                 ; eventuelle ERR#
F480-   60          RTS                 ; mit Carry gelöscht/gesetzt von der RWTB
```

## $CC – CLOSE COMMAND ; Vorher nur internes SET TIME

```
F481-   A0 01       LDY   #$01
F483-   B1 40       LDA   ($40),Y       ; File-Refno aus dem „user"-Parameterblock
F485-   D0 35       BNE   $F4BC         ; CLOSE SPECIFIC
F487-   8D BE FE    STA   $FEBE         ; Refno = $00 „ALL": Reset des ERR#-Speicher
F48A-   A9 00       LDA   #$00          ; unnötig

F48C-   8D 92 FE    STA   $FE92         ; Setzen des FCB-Index
F48F-   A8          TAY
F490-   B9 1B D8    LDA   $D81B,Y       ; „LEVEL" im FCB-Eintrag (s. „Spezielles")
F493-   CD 94 BF    CMP   $BF94
F496-   90 15       BCC   $F4AD         ; LEVEL auf der Global Page kleiner: NEXT
F498-   B9 00 D8    LDA   $D800,Y       ; gültiger FCB-Eintrag?
F49B-   F0 10       BEQ   $F4AD         ; nein, NEXT
F49D-   20 1E F5    JSR   $F51E         ; FLUSH SPECIFIC für diesen Eintrag
F4A0-   B0 4D       BCS   $F4EF
F4A2-   20 C1 F4    JSR   $F4C1         ; CLOSE SPECIFIC für diesen Eintrag
F4A5-   A0 01       LDY   #$01
F4A7-   B1 40       LDA   ($40),Y       ; ist Refno = $00 (ALL)?
F4A9-   F0 02       BEQ   $F4AD         ; ja, NEXT
F4AB-   B0 42       BCS   $F4EF
F4AD-   AD 92 FE    LDA   $FE92         ; FCB-Index + $20
F4B0-   18          CLC
F4B1-   69 20       ADC   #$20
F4B3-   90 D7       BCC   $F48C         ; Loop bis zum Ende des FCB
F4B5-   AD BE FE    LDA   $FEBE         ; Zwischenspeicher für ERR#
F4B8-   D0 01       BNE   $F4BB         ; RTS mit Carry set (von $E6A9) bei Fehlern
F4BA-   18          CLC
F4BB-   60          RTS
```

## – CLOSE SPECIFIC – Einsprung für CLOSE eines bestimmten Files

```
F4BC-   20 26 F5    JSR   $F526         ; FLUSH SPECIFIC – bestimmtes File
F4BF-   B0 2E       BCS   $F4EF
```

Xref: $F4A2 – CLOSE ALL Loop

```
F4C1-   AC 92 FE    LDY   $FE92         ; FCB-Index
F4C4-   B9 0B D8    LDA   $D80B,Y       ; Puffer-Index von der System Global Page
F4C7-   20 4A FC    JSR   $FC4A         ; DEALLOCATE BUFFER
F4CA-   B0 23       BCS   $F4EF
F4CC-   A9 00       LDA   #$00
F4CE-   AC 92 FE    LDY   $FE92
F4D1-   99 00 D8    STA   $D800,Y       ; der FCB-Eintrag wird ungültig gesetzt
F4D4-   B9 01 D8    LDA   $D801,Y
F4D7-   8D 30 BF    STA   $BF30         ; Unitnummer
F4DA-   20 76 E8    JSR   $E876         ; sucht den VCB-Eintrag für das File
F4DD-   AE 91 FE    LDX   $FE91         ; FCB-Index, von $DA69 gesetzt
F4E0-   DE 1E D9    DEC   $D91E,X       ; Anzahl der OPEN Files auf dem Volume
F4E3-   D0 08       BNE   $F4ED
F4E5-   BD 11 D9    LDA   $D911,X       ; keine OPEN Files mehr auf dem Volume:
F4E8-   29 7F       AND   #$7F          ; Reset des Flags „OPEN File(s)"
```

```
F4EA-   9D 11 D9    STA   $D911,X
F4ED-   18          CLC
F4EE-   60          RTS                 ; => $DE78 oder CLOSE ALL Loop
```

– Error-„Verlängerung"

```
F4EF-   4C F7 F5    JMP   $F5F7         ; zum Error Handler für CLOSE und FLUSH
```

**$CD – FLUSH COMMAND** ; Vorher nur internes SET TIME

```
F4F2-   A0 01       LDY   #$01          ; Reference Number: Parameter Nummer 1
F4F4-   B1 40       LDA   ($40),Y       ; aus dem „user"-Parameterblock
F4F6-   D0 2E       BNE   $F526         ; FLUSH SPECIFIC
F4F8-   8D BE FE    STA   $FEBE         ; $00 = „ALL", => Zwischenspeicher für ERR#
F4FB-   A9 00       LDA   #$00          ; unnötig

F4FD-   8D 92 FE    STA   $FE92         ; FCB-Index
F500-   A8          TAY
F501-   B9 00 D8    LDA   $D800,Y
F504-   F0 05       BEQ   $F50B         ; ungültiger Eintrag: NEXT
F506-   20 1E F5    JSR   $F51E         ; FLUSH SPECIFIC
F509-   B0 10       BCS   $F51B
F50B-   AD 92 FE    LDA   $FE92         ; FCB-Index + $20 (NEXT)
F50E-   18          CLC
F50F-   69 20       ADC   #$20
F511-   90 EA       BCC   $F4FD
F513-   18          CLC
F514-   AD BE FE    LDA   $FEBE         ; Zwischenspeicher für ERR#
F517-   F0 01       BEQ   $F51A         ; keine Fehler während des Loops
F519-   38          SEC
F51A-   60          RTS
```

– Error-„Verlängerung"

```
F51B-   4C F7 F5    JMP   $F5F7         ; zum Error Handler für CLOSE und FLUSH
```

**– FLUSH SPECIFIC**

Xref: $F49D – CLOSE ALL Loop
$F506 – FLUSH ALL Loop

```
F51E-   20 EB E1    JSR   $E1EB         ; Teil von „Set File Data" – setzt Puffer,
F521-   90 0D       BCC   $F530                   Pointer etc.
F523-   4C F7 F5    JMP   $F5F7         ; Error Handler
```

– Einsprung für FLUSH eines bestimmten Files

Xref: $F4BC – CLOSE eines bestimmten Files
$F66D – SET EOF: der neue EOF ist größer als der alte EOF
$F752 – SET EOF: nach Verkürzung des Files („eingebaut" seit Ver. 1.0.2)

```
F526-   A9 00       LDA   #$00
F528-   8D BE FE    STA   $FEBE         ; Zwischenspeicher für ERR#
F52B-   20 D0 E1    JSR   $E1D0         ; „Set File Data" mit Test der Refno
F52E-   B0 EB       BCS   $F51B
```

```
F530-  B9 09 D8   LDA  $D809,Y      ; ACCESS
F533-  29 02      AND  #$02         ; ist WRITE erlaubt?
F535-  F0 DC      BEQ  $F513        ; nein, CLC/RTS
F537-  B9 1C D8   LDA  $D81C,Y      ; „File altered"?
F53A-  30 07      BMI  $F543        ; ja, FLUSH „always"
F53C-  20 06 F6   JSR  $F606        ; GET STATUS: wenn Bit 4 („ACTIVATEd") und
F53F-  29 70      AND  #$70           Bit 6 („Datablock altered") beide gelöscht
F541-  F0 D0      BEQ  $F513          sind => kein WRITE => CLC/RTS

F543-  20 06 F6   JSR  $F606        ; GET STATUS
F546-  29 40      AND  #$40         ; „Datablock altered"?
F548-  F0 05      BEQ  $F54F
F54A-  20 94 EE   JSR  $EE94        ; ja, Rewrite des Datenblocks (Filepuffer)
F54D-  B0 CC      BCS  $F51B
F54F-  20 06 F6   JSR  $F606        ; GET STATUS
F552-  29 80      AND  #$80         ; „Indexblock altered"?
F554-  F0 05      BEQ  $F55B
F556-  20 A8 EE   JSR  $EEA8        ; Ja, Rewrite des Indexblocks (Filepuffer)
F559-  B0 C0      BCS  $F51B
```

– „Zusammenbau" des File Info für das Neuschreiben des File-Eintrags in ENTRY & UPDATE

```
F55B-  A9 06      LDA  #$06
F55D-  AA         TAX
F55E-  0D 92 FE   ORA  $FE92        ; FCB-Index
F561-  A8         TAY
F562-  B9 00 D8   LDA  $D800,Y      ; Schritt 1: Kopie aus dem FCB-Eintrag
F565-  9D 58 FE   STA  $FE58,X        => $FE59:    Unitnummer
F568-  88         DEY                 => $FE5A/5B: Key Blockno der Directory
F569-  CA         DEX                 => $FE5C/5D: Nummer des DIR-Blocks
F56A-  D0 F6      BNE  $F562          => $FE5E:    Eintragsnummer
F56C-  8D 30 BF   STA  $BF30        ; Unitnummer
F56F-  AD 5A FE   LDA  $FE5A        ; Schritt 2: Laden des Key Blocks der
F572-  85 46      STA  $46            Directory, in dem der File steht
F574-  AD 5B FE   LDA  $FE5B
F577-  85 47      STA  $47          ; Blocknummer gesetzt,
F579-  20 EE EB   JSR  $EBEE        ; Lesen des Key Blocks auf $DC00/$DD00
F57C-  B0 9D      BCS  $F51B
F57E-  20 AB E6   JSR  $E6AB        ; => $FE51: Eintragslänge; $FE46..:Parent Info
F581-  AD 5C FE   LDA  $FE5C
F584-  AC 5D FE   LDY  $FE5D
F587-  CD 5A FE   CMP  $FE5A        ; wenn der Eintrag nicht im Key Block
F58A-  D0 05      BNE  $F591          selber steht, muß jetzt der entsprechende
F58C-  CC 5B FE   CPY  $FE5B          Block („Parent Block") gelesen werden
F58F-  F0 07      BEQ  $F598
F591-  85 46      STA  $46
F593-  84 47      STY  $47
F595-  20 EE EB   JSR  $EBEE        ; Lesen des „Parent Block"

F598-  20 9D E4   JSR  $E49D        ; Schritt 3: Bestimmung des Eintrags
F59B-  20 A8 E5   JSR  $E5A8        ; und Herauskopieren der gesamten Daten
F59E-  AC 92 FE   LDY  $FE92          auf $FE5F...
F5A1-  B9 18 D8   LDA  $D818,Y
F5A4-  8D 72 FE   STA  $FE72        ; Überschreiben von BLOCKS USED
F5A7-  B9 19 D8   LDA  $D819,Y
F5AA-  8D 73 FE   STA  $FE73
F5AD-  A2 00      LDX  #$00
F5AF-  B9 15 D8   LDA  $D815,Y      ; Überschreiben des EOF
F5B2-  9D 74 FE   STA  $FE74,X
F5B5-  E8         INX
F5B6-  E0 03      CPX  #$03
F5B8-  F0 09      BEQ  $F5C3
```

```
F5BA-   B9 0C D8    LDA   $D80C,Y
F5BD-   9D 6F FE    STA   $FE6F,X       ; Überschreiben der Nummer des
F5C0-   C8          INY                   „ersten Blocks im File"
F5C1-   D0 EC       BNE   $F5AF
F5C3-   B9 05 D8    LDA   $D805,Y       ; = „$D807,Y", d.h. Storage Type
F5C6-   0A          ASL
F5C7-   0A          ASL
F5C8-   0A          ASL
F5C9-   0A          ASL
F5CA-   8D 86 FE    STA   $FE86         ; Scratch
F5CD-   AD 5F FE    LDA   $FE5F         ; Storage Type  &  Namenslänge aus der DIR
F5D0-   29 0F       AND   #$0F          ; = Namenslänge
F5D2-   0D 86 FE    ORA   $FE86         ; + (eventuell veränderter Storage Type)
F5D5-   8D 5F FE    STA   $FE5F
F5D8-   20 B6 E4    JSR   $E4B6         ; ENTRY & UPDATE
F5DB-   B0 1A       BCS   $F5F7
F5DD-   AC 92 FE    LDY   $FE92         ; FCB-Index
F5E0-   B9 1C D8    LDA   $D81C,Y       ; Zurücksetzen von „File altered"
F5E3-   29 7F       AND   #$7F
F5E5-   99 1C D8    STA   $D81C,Y
F5E8-   AD 59 FE    LDA   $FE59         ; Unitnummer des Files
F5EB-   CD A6 FE    CMP   $FEA6         ; Unitnummer des „momentanen" VBM-Blocks
F5EE-   D0 05       BNE   $F5F5
F5F0-   20 93 EB    JSR   $EB93         ; Rewrite, falls noch ein veränderter
F5F3-   B0 02       BCS   $F5F7           VBM-Block dieses Units im Speicher
F5F5-   18          CLC
F5F6-   60          RTS
```

– Error Handler für CLOSE UND FLUSH: Abfangen von Fehlern für die „ALL"-Loops

Xref: $F4EF – JP von der „Verlängerung" bei CLOSE
$F51B – JP von der „Verlängerung" bei FLUSH
$F523 – JP von FLUSH SPECIFIC

```
F5F7-   A0 01       LDY   #$01
F5F9-   48          PHA                 ; ERR#
F5FA-   B1 40       LDA   ($40),Y       ; Refno aus dem „user"-Parameterblock
F5FC-   D0 06       BNE   $F604         ; war nicht $00 („ALL"): RTS mit Carry gesetzt
F5FE-   18          CLC
F5FF-   68          PLA                 ; ansonsten wird die ERR# gespeichert,
F600-   8D BE FE    STA   $FEBE           RTS mit Carry gelöscht
F603-   60          RTS
F604-   68          PLA
F605-   60          RTS
```

– **GET STATUS** des „momentanen" Files

Xref: $F06D/$F2B5/$F2CD/$F2E1/$F2F2/$F380/$F43D
$F458/$F53C/$F543/$F54F

```
F606-   AC 92 FE    LDY   $FE92         ; FCB-Index
F609-   B9 08 D8    LDA   $D808,Y       ; STATUS im FCB-Eintrag
F60C-   60          RTS
```

– Error Exit von SET EOF

```
F60D-   A9 4E       LDA   #$4E          ; ERR#4E: FILE ACCESS ERROR (FILE LOCKED)
F60F-   38          SEC
F610-   60          RTS
```

**$D0 – SET EOF COMMAND** ; File muß OPEN sein – Set File Data vorher

```
F611-   20 18 F2    JSR   $F218         ; holt Storage Type des Files
F614-   C9 04       CMP   #$04          ; SET EOF geht nicht für (Sub)directories
F616-   B0 F5       BCS   $F60D
F618-   0A          ASL
F619-   0A          ASL
F61A-   0A          ASL
F61B-   0A          ASL
F61C-   8D C1 FE    STA   $FEC1         ; Storage Type ins höhere Nibble
F61F-   B9 09 D8    LDA   $D809,Y       ; ACCESS (Y-Index von $F218)
F622-   29 02       AND   #$02
F624-   F0 E7       BEQ   $F60D         ; WRITE nicht erlaubt: ACCESS ERROR
F626-   20 58 F4    JSR   $F458         ; „ACTIVATE" – Test auf „Ready" und WPROT
F629-   B0 E2       BCS   $F60D         ; nicht „Ready" oder schreibgeschützt
F62B-   AC 92 FE    LDY   $FE92         ; FCB-Index
F62E-   C8          INY
F62F-   C8          INY
F630-   A2 02       LDX   #$02
F632-   B9 15 D8    LDA   $D815,Y       ; der EOF wird auf $FE8A/8B/8C kopiert
F635-   9D 8A FE    STA   $FE8A,X
F638-   88          DEY
F639-   CA          DEX
F63A-   10 F6       BPL   $F632
F63C-   A0 04       LDY   #$04          ; Parameter $04/$05/$06 aus dem
F63E-   A2 02       LDX   #$02            Block des „user"-Programms: EOF gewünscht
F640-   B1 40       LDA   ($40),Y
F642-   9D AA FE    STA   $FEAA,X       ; wird auf $FEAA/AB/AC kopiert
F645-   88          DEY
F646-   CA          DEX
F647-   10 F7       BPL   $F640
F649-   A2 02       LDX   #$02
F64B-   BD 8A FE    LDA   $FE8A,X       ; Vergleich „EOF jetzt" ($FE8A..)
F64E-   DD AA FE    CMP   $FEAA,X       ; mit „EOF gewünscht" ($FEAA..)
F651-   90 05       BCC   $F658
F653-   D0 18       BNE   $F66D         ; wenn der „EOF gewünscht" größer als der
F655-   CA          DEX                   „EOF jetzt" ist, wird der EOF schlicht
F656-   10 F3       BPL   $F64B           via $F658 neu gesetzt, danach RTS => $DE78
```

– Setzen des neuen EOF im FCB-Eintrag

Xref: $F651 bzw. $F656 – neuer EOF ist größer
$F74F – SET EOF (Verkürzung des Files) erledigt

```
F658-   A0 04       LDY   #$04
F65A-   AE 92 FE    LDX   $FE92         ; FCB-Index
F65D-   E8          INX
F65E-   E8          INX
F65F-   B1 40       LDA   ($40),Y       ; der „EOF gewünscht" aus dem Parameter-
F661-   9D 15 D8    STA   $D815,X         block des „user"-Programms wird in
F664-   CA          DEX                   den FCB-Eintrag kopiert
F665-   88          DEY
F666-   C0 02       CPY   #$02
F668-   B0 F5       BCS   $F65F
F66A-   4C 66 FA    JMP   $FA66         ; setzt „File altered"(STATUS): => $DE78
```

– EOF „gewünscht" < EOF „jetzt": VERKÜRZUNG DES FILES

```
F66D-   20 26 F5    JSR   $F526         ; FLUSH des Files – falls noch geänderte
F670-   B0 9E       BCS   $F610           Daten/Indexblocks im Filepuffer
F672-   A6 4B       LDX   $4B           ; SET MARK (FLUSH Exit) hat Daten-Pointer
F674-   E8          INX                   auf die „Position jetzt" (Seitengrenze)
```

```
F675-  E8          INX                 gesetzt: Reset auf Anfang des Datenteils,
F676-  86 49       STX  $49            d.h. 2 Seiten oberhalb des Indexteils
F678-  A6 4A       LDX  $4A          ; lo-Bytes Indexteil-Pointer =>
F67A-  86 48       STX  $48          ; Datenteil-Pointer
F67C-  AC 92 FE    LDY  $FE92        ; FCB-Index
F67F-  C8          INY
F680-  C8          INY
F681-  A2 02       LDX  #$02
F683-  B9 12 D8    LDA  $D812,Y      ; Vergleich „EOF gewünscht" mit
F686-  DD AA FE    CMP  $FEAA,X      ; „Position jetzt"
F689-  90 17       BCC  $F6A2
F68B-  D0 04       BNE  $F691
F68D-  88          DEY
F68E-  CA          DEX
F68F-  10 F2       BPL  $F683
F691-  AC 92 FE    LDY  $FE92        ; FCB-Index
F694-  A2 00       LDX  #$00         ; wenn „Position jetzt" > „EOF gewünscht",
F696-  BD AA FE    LDA  $FEAA,X        wird die „Position jetzt" im FCB-Eintrag
F699-  99 12 D8    STA  $D812,Y        direkt auf den neuen EOF gesetzt
F69C-  C8          INY
F69D-  E8          INX
F69E-  E0 03       CPX  #$03
F6A0-  D0 F4       BNE  $F696

F6A2-  AD AA FE    LDA  $FEAA        ; „EOF gewünscht" – lobyte
F6A5-  8D C6 FE    STA  $FEC6
F6A8-  AD AB FE    LDA  $FEAB        ; mittleres Byte
F6AB-  8D C5 FE    STA  $FEC5
F6AE-  29 01       AND  #$01         ; ergibt $FEC6/C7 = $00, wenn auf Blockgrenze
F6B0-  8D C7 FE    STA  $FEC7
F6B3-  AD AC FE    LDA  $FEAC
F6B6-  4A          LSR
F6B7-  8D C4 FE    STA  $FEC4        ; UNTERE GRENZE IM MASTER-BLOCK
F6BA-  6E C5 FE    ROR  $FEC5        ; UNTERE GRENZE IM INDEX-BLOCK
F6BD-  AD C6 FE    LDA  $FEC6
F6C0-  0D C7 FE    ORA  $FEC7
F6C3-  D0 1D       BNE  $F6E2        ; Sprung, wenn nicht auf Blockgrenze
F6C5-  AD C5 FE    LDA  $FEC5        ; auf Blockgrenze: Korrektur der
F6C8-  38          SEC                 Untergrenzen. Beispiel: EOF = $400 =>
F6C9-  E9 01       SBC  #$01           es bleiben nur Blocks 00 und 01 übrig,
F6CB-  8D C5 FE    STA  $FEC5          Block 02 wird dann nicht benötigt
F6CE-  A9 02       LDA  #$02
F6D0-  B0 0D       BCS  $F6DF
F6D2-  CE C4 FE    DEC  $FEC4
F6D5-  10 08       BPL  $F6DF
F6D7-  A9 00       LDA  #$00         ; führt diese Korrektur zu einem Ergebnis
F6D9-  8D C4 FE    STA  $FEC4          kleiner Null, werden die Grenzen auf
F6DC-  8D C5 FE    STA  $FEC5          00 00 00 gesetzt.
F6DF-  8D C7 FE    STA  $FEC7

F6E2-  AC 92 FE    LDY  $FE92        ; FCB-Index
F6E5-  B9 0C D8    LDA  $D80C,Y      ; wenn SET EOF die File-Länge kürzt, wird
F6E8-  8D BF FE    STA  $FEBF          dabei eventuell der Storage Type und
F6EB-  B9 0D D8    LDA  $D80D,Y        die Nummer „erster Block" geändert.
F6EE-  8D C0 FE    STA  $FEC0        ; $FEBF/$FEC0: „erster Block im File"
F6F1-  A9 00       LDA  #$00
F6F3-  8D C2 FE    STA  $FEC2        ; $FEC2/C3: Zähler für entfernte Blocks
F6F6-  8D C3 FE    STA  $FEC3
F6F9-  20 78 FA    JSR  $FA78        ; REMOVE BLOCKS
F6FC-  08          PHP
F6FD-  48          PHA               ; eventuelle ERR#
F6FE-  38          SEC
F6FF-  AC 92 FE    LDY  $FE92        ; FCB-Index
F702-  A2 00       LDX  #$00
```

```
F704-  BD BF FE   LDA  $FEBF,X      ; Update der Nummer „erster Block" im FCB
F707-  99 0C D8   STA  $D80C,Y
F70A-  B9 18 D8   LDA  $D818,Y      ; BLOCKS USED
F70D-  FD C2 FE   SBC  $FEC2,X      ; minus Anzahl der entfernten Blocks
F710-  99 18 D8   STA  $D818,Y
F713-  C8         INY
F714-  E8         INX
F715-  8A         TXA
F716-  29 01      AND  #$01         ; Carry nicht verändern – 16-Bit Subtraktion
F718-  D0 EA      BNE  $F704

F71A-  AD C1 FE   LDA  $FEC1        ; Storage Type des Files
F71D-  4A         LSR
F71E-  4A         LSR
F71F-  4A         LSR
F720-  4A         LSR               ; zurück ins untere Nibble (s. $F618)
F721-  AC 92 FE   LDY  $FE92
F724-  99 07 D8   STA  $D807,Y      ; Update im FCB
F727-  20 AF ED   JSR  $EDAF        ; löscht die Erweiterungsbits in STATUS
F72A-  20 F3 F9   JSR  $F9F3        ; erhöht FREE BLOCKS auf dem Volume um
F72D-  AC 92 FE   LDY  $FE92          die Anzahl der entfernten Blocks
F730-  C8         INY
F731-  C8         INY
F732-  A2 02      LDX  #$02
F734-  B9 12 D8   LDA  $D812,Y      ; Kopie der „Position jetzt" aus dem FCB
F737-  9D AA FE   STA  $FEAA,X      ; als „Position gewünscht"
F73A-  49 80      EOR  #$80
F73C-  99 12 D8   STA  $D812,Y      ; „Verdrehen" und Zurückschreiben:
F73F-  88         DEY                 Folge: der nächste Aufruf von SET MARK
F740-  CA         DEX                 erzwingt ein neues Einlesen in den
F741-  10 F1      BPL  $F734          Filepuffer
F743-  20 48 EC   JSR  $EC48        ; SET MARK
F746-  90 07      BCC  $F74F
F748-  AA         TAX
F749-  68         PLA               ; Fehler: das hier sind noch ERR# und P-Reg
F74A-  28         PLP                 von REMOVE BLOCKS
F74B-  8A         TXA               ; ERR# wird gegen die ERR# von SET MARK
F74C-  38         SEC                 ausgewechselt. Passiert sowohl bei REMOVE
F74D-  08         PHP                 BLOCKS als auch bei SET MARK ein Fehler,
F74E-  48         PHA                 wird der SET MARK-Fehler  zurückgebracht
F74F-  20 58 F6   JSR  $F658        ; setzt den EOF im FCB-Eintrag neu

F752-  20 26 F5   JSR  $F526        ; FLUSH – Update des File-Eintrags / VBM
F755-  90 07      BCC  $F75E        ; ok. Exit
F757-  AA         TAX               ; ERR# von FLUSH
F758-  68         PLA               ; ERR# v. REMOVE BLOCKS bzw. SET MARK
F759-  28         PLP                 wird vom Stack geholt und die ERR#
F75A-  8A         TXA                (+ Carry) von FLUSH eingesetzt.
F75B-  38         SEC
F75C-  08         PHP               Dieser Aufruf (+ Wechsel der ERR#) ist seit
F75D-  48         PHA               der Version 1.0.2 eingebaut

F75E-  68         PLA
F75F-  28         PLP
F760-  60         RTS               ; => $DE78
```

## $D1 – GET EOF COMMAND ; File muß OPEN sein – Set File Data vorher

```
F761-  AE 92 FE   LDX  $FE92        ; FCB-Index, gesetzt von Set File Data
F764-  A0 02      LDY  #$02
F766-  BD 15 D8   LDA  $D815,X      ; der EOF aus dem FCB-Eintrag wird in
F769-  91 40      STA  ($40),Y        den Parameterblock des „user"-Programms
```

```
F76B-   E8          INX                 hineinkopiert
F76C-   C8          INY
F76D-   C0 05       CPY   #$05
F76F-   D0 F5       BNE   $F766
F771-   18          CLC
F772-   60          RTS                 ; => $DE78
```

**$C9 – NEWLINE COMMAND** ; File muß OPEN sein – Set File Data vorher

```
F773-   A0 02       LDY   #$02          ; Byte $02 aus dem „user"-Parameterblock:
F775-   B1 40       LDA   ($40),Y       ; NEWLINE AND MASK
F777-   AE 92 FE    LDX   $FE92         ; FCB-Index, von Set File Data gesetzt
F77A-   9D 1F D8    STA   $D81F,X       ; FCB-Eintrag, Byte $1F: AND MASK
F77D-   C8          INY
F77E-   B1 40       LDA   ($40),Y       ; Byte $03:
F780-   9D 0A D8    STA   $D80A,X       ; FCB-Eintrag, Byte $0A: NEWLINE CHAR
F783-   18          CLC
F784-   60          RTS                 ; => $DE78
```

**$C4 – GET FILE INFO COMMAND** ; Vorher Copy & Verify Pathname

```
F785-   20 A3 E5    JSR   $E5A3         ; GET FILE – sucht & lädt File-Eintrag
F788-   90 42       BCC   $F7CC
F78A-   C9 40       CMP   #$40          ; INVALID PATHNAME, d.h. Ende in VolDIR?
F78C-   38          SEC
F78D-   D0 5A       BNE   $F7E9         ; andere Fehler werden nicht abgefangen
```

– Volume Directory

```
F78F-   A9 F0       LDA   #$F0          ; der von GET FILE „getürkte" Storage Type
F791-   8D 5F FE    STA   $FE5F           wird wieder auf „VolDIR" gesetzt
F794-   A9 00       LDA   #$00
F796-   8D 94 FE    STA   $FE94         ; BLOCKS NEEDED auf 00 00
F799-   8D 95 FE    STA   $FE95
F79C-   AE 91 FE    LDX   $FE91         ; VCB-Index, gesetzt von GET FILE
F79F-   20 7E E9    JSR   $E97E         ; SPACE ON VOLUME
F7A2-   AE 91 FE    LDX   $FE91
F7A5-   BD 15 D9    LDA   $D915,X       ; von SPACE ON VOLUME im VCB gesetzte
F7A8-   8D 95 FE    STA   $FE95           Anzahl der freien Blocks
F7AB-   BD 14 D9    LDA   $D914,X       ; Lobyte
F7AE-   8D 94 FE    STA   $FE94
F7B1-   BD 13 D9    LDA   $D913,X       ; TOTAL BLOCKS (Gesamtzahl) gehen in das
F7B4-   8D 7F FE    STA   $FE7F         ; AUX-TYPE – Field des FILE INFO
F7B7-   48          PHA
F7B8-   BD 12 D9    LDA   $D912,X
F7BB-   8D 7E FE    STA   $FE7E         ; Lobyte
F7BE-   38          SEC
F7BF-   ED 94 FE    SBC   $FE94         ; TOTAL BLOCKS minus BLOCKS FREE ergibt
F7C2-   8D 72 FE    STA   $FE72         ; BLOCKS USED, Ergebnis in BLOCKS-USED
F7C5-   68          PLA                   des File Info
F7C6-   ED 95 FE    SBC   $FE95
F7C9-   8D 73 FE    STA   $FE73
```

– Subdirectories/Datenfiles

```
F7CC-   AD 5F FE    LDA   $FE5F         ; Storage Type
F7CF-   4A          LSR
F7D0-   4A          LSR
F7D1-   4A          LSR
```

```
F7D2-  4A          LSR
F7D3-  8D 5F FE    STA   $FE5F        ; die Namenslänge wird ignoriert
F7D6-  A0 11       LDY   #$11
F7D8-  B9 0C FE    LDA   $FE0C,Y      ; $10/$1F/$20/$00/$13/$14/$21/$22/$23/$24
F7DB-  29 7F       AND   #$7F                        **  **  **
F7DD-  AA          TAX
F7DE-  BD 5F FE    LDA   $FE5F,X      ; Kopie des Parametersatzes in den
F7E1-  91 40       STA   ($40),Y      ; Block des „user"-Programms
F7E3-  88          DEY                  die mit ** markierten Bytes haben
F7E4-  C0 03       CPY   #$03           zusätzlich noch Bit 7 gesetzt
F7E6-  B0 F0       BCS   $F7D8          (s. SET FILE INFO)
F7E8-  18          CLC
F7E9-  60          RTS                ; => $DE78
```

## $C3 – SET FILE INFO COMMAND ; Copy & Verify Pathname/SET TIME vorher

```
F7EA-  20 A3 E5    JSR   $E5A3        ; GET FILE
F7ED-  B0 25       BCS   $F814
F7EF-  AD 95 BF    LDA   $BF95        ; BACKUP BIT
F7F2-  49 20       EOR   #$20         ; wird für ENTRY & UPDATE nur gesetzt, wenn
F7F4-  2D 7D FE    AND   $FE7D        ; ACCCESS des Files das BUBIT gesetzt
F7F7-  29 20       AND   #$20           hat und BACKUP BIT auf der Global Page
F7F9-  8D B4 FE    STA   $FEB4          nicht gesetzt ist (s. „Spezielles")
F7FC-  A0 0D       LDY   #$0D
F7FE-  BE 0C FE    LDX   $FE0C,Y      ; Adress-Übersetzungstabelle. Hier werden
F801-  30 05       BMI   $F808          die drei Bytes des NULL-FIELD über-
F803-  B1 40       LDA   ($40),Y        sprungen, die anderen Parameter aus dem
F805-  9D 5F FE    STA   $FE5F,X        „user"-Block werden in das File Info
F808-  88          DEY                  kopiert (s. GET FILE INFO).
F809-  C0 03       CPY   #$03           Es erfolgt keine Prüfung, außer für
F80B-  B0 F1       BCS   $F7FE          den Storage Type !!
F80D-  29 1C       AND   #$1C         ; letztes Byte: Storage Type + Namenslänge
F80F-  F0 04       BEQ   $F815        ; Sprung für Storage Type 0(!)1/2/3
F811-  A9 4E       LDA   #$4E         ; ERR=4E: UNSUPPORTED STORAGE TYPE
F813-  38          SEC
F814-  60          RTS
F815-  A0 0B       LDY   #$0B         ; Parameter $0B.. aus dem „user"-Parmblock:
F817-  B1 40       LDA   ($40),Y      ; MODIFICATION DATE & TIME
F819-  F0 03       BEQ   $F81E
F81B-  4C C6 E4    JMP   $E4C6        ; spezifiziert: ENTRY & UPDATE ohne Sys Time
F81E-  4C B6 E4    JMP   $E4B6        ; ENTRY & UPDATE mit System Time
```

## $C2 – RENAME COMMAND ; Copy & Verify Pathname vorher

```
F821-  20 B6 E5    JSR   $E5B6        ; FIND FILE
F824-  90 3D       BCC   $F863
F826-  C9 40       CMP   #$40         ; INVALID PATHNAME, d.h. Volume Directory?
F828-  D0 18       BNE   $F842        ; andere Fehler werden nicht maskiert
```

### – RENAME für Volumes

```
F82A-  20 4B F9    JSR   $F94B        ; setzt Pointer auf 2. Namen, Copy & Verify
F82D-  B0 13       BCS   $F842        ; SEC/RTS (INVALID PATHNAME)
F82F-  AC 00 D7    LDY   $D700        ; Länge des ersten Namens im neuen Path
F832-  C8          INY
F833-  B9 00 D7    LDA   $D700,Y      ; die Länge des zweiten Namens im neuen

F836-  D0 7F       BNE   $F8B7          Path muß Null sein, d.h. nur ein Name
F838-  AE 91 FE    LDX   $FE91        ; VCB-Index, gesetzt von FIND FILE
F83B-  BD 11 D9    LDA   $D911,X      ; OPEN File(s) auf dem Volume?
F83E-  10 04       BPL   $F844
```

```
F840-  A9 50      LDA  #$50        ; ERR#50: FILE IS OPEN
F842-  38         SEC
F843-  60         RTS

F844-  A0 00      LDY  #$00
F846-  B9 00 D7   LDA  $D700,Y
F849-  09 F0      ORA  #$F0
F84B-  20 3C F9   JSR  $F93C       ; kopiert den neuen Namen und schreibt
F84E-  B0 69      BCS  $F8B9         den geladenen VolDIR Key Block zurück
F850-  A0 00      LDY  #$00
F852-  AE 91 FE   LDX  $FE91

F855-  B9 00 D7   LDA  $D700,Y     ; danach wird der neue Name in den
F858-  F0 07      BEQ  $F861         FCB-Eintrag kopiert
F85A-  9D 00 D9   STA  $D900,X
F85D-  C8         INY
F85E-  E8         INX
F85F-  D0 F4      BNE  $F855
F861-  18         CLC
F862-  60         RTS              ; => $DE78
```

– RENAME für Subdirectories/Datenfiles

```
F863-  20 59 F9   JSR  $F959       ; holt den Startindex des alten Pathname
F866-  B9 00 D7   LDA  $D700,Y
F869-  9D 00 DC   STA  $DC00,X     ; der alte Pathname wird in den Directory
F86C-  30 04      BMI  $F872         Space kopiert
F86E-  C8         INY
F86F-  E8         INX
F870-  D0 F4      BNE  $F866

F872-  20 4B F9   JSR  $F94B       ; Pointer auf neuen Pathname, Copy & Verify
F875-  B0 42      BCS  $F8B9
F877-  20 59 F9   JSR  $F959       ; holt jetzt Startindex des neuen Pathname
F87A-  B9 00 D7   LDA  $D700,Y
F87D-  DD 00 DC   CMP  $DC00,X     ; Vergleich beider Paths, bis ein unter-
F880-  08         PHP                schiedlicher Name gefunden wird.

F881-  29 F0      AND  #$F0        ; wenn das $00 ergibt, haben wir ein
F883-  D0 06      BNE  $F88B         Längenbyte, d.h. den nächsten Namen
F885-  8C B9 FE   STY  $FEB9         im Path. Die Start-Indizes der jeweiligen
F888-  8E BA FE   STX  $FEBA         Namen im Path werden gespeichert
F88B-  28         PLP              ; vom Vergleich
F88C-  D0 09      BNE  $F897

F88E-  E8         INX
F88F-  C8         INY
F890-  B9 00 D7   LDA  $D700,Y
F893-  D0 E8      BNE  $F87D
F895-  18         CLC              ; beide Paths total gleich: RTS ohne Fehler
F896-  60         RTS
```

– verschiedene Namen in beiden Paths gefunden

```
F897-  AC B9 FE   LDY  $FEB9       ; Start-Index des verschiedenen Namens im

F89A-  B9 00 D7   LDA  $D700,Y       neuen Path: + Namenslänge => Index zum
F89D-  38         SEC                Längenbyte des darauffolgenden Namens
F89E-  6D B9 FE   ADC  $FEB9
F8A1-  A8         TAY
F8A2-  B9 00 D7   LDA  $D700,Y     ; das Längenbyte des nächsten Namens muß 00
F8A5-  D0 10      BNE  $F8B7         sein, d.h. der Unterschied zwischen den

F8A7-  AE BA FE   LDX  $FEBA         beiden Paths muß im letzten Namen
F8AA-  BD 00 DC   LDA  $DC00,X       auftreten!
F8AD-  38         SEC
```

```
F8AE-  6D BA FE   ADC  $FEBA        ; derselbe Test für den alten Pathnamen
F8B1-  AA         TAX
F8B2-  BD 00 DC   LDA  $DC00,X
F8B5-  F0 04      BEQ  $F8BB
F8B7-  A9 40      LDA  #$40         ; ERR#40: INVALID PATHNAME
F8B9-  38         SEC
F8BA-  60         RTS

F8BB-  20 B6 E5   JSR  $E5B6        ; FIND FILE mit dem neuen Pathnamen
F8BE-  B0 04      BCS  $F8C4
F8C0-  A9 47      LDA  #$47         ; ERR#47: DUPLICATE FILENAME
F8C2-  38         SEC                 (File mit diesem Namen existiert bereits)
F8C3-  60         RTS
F8C4-  C9 46      CMP  #$46         ; war der Fehler „FILE NOT FOUND"?
F8C6-  D0 F1      BNE  $F8B9        ; nein, andere Fehler werden nicht maskiert
F8C8-  20 8A E0   JSR  $E08A        ; Copy & Verify Pathname holt den alten Path
F8CB-  20 A3 E5   JSR  $E5A3        ; GET FILE
F8CE-  B0 E9      BCS  $F8B9
F8D0-  20 B3 EF   JSR  $EFB3        ; sucht den FCB nach OPEN Files ab
F8D3-  A9 50      LDA  #$50         ; ERR#50: FILE IS OPEN
F8D5-  B0 E2      BCS  $F8B9        ; $EFB3 mit SEC, wenn File im FCB gefunden
F8D7-  AD 7D FE   LDA  $FE7D        ; ACCESS
F8DA-  29 40      AND  #$40         ; RENAME erlaubt?
F8DC-  D0 04      BNE  $F8E2
F8DE-  A9 4E      LDA  #$4E         ; ERR#4E: FILE ACCESS ERROR
F8E0-  38         SEC
F8E1-  60         RTS
F8E2-  AD 5F FE   LDA  $FE5F        ; Storage Type
F8E5-  29 F0      AND  #$F0
F8E7-  C9 D0      CMP  #$D0
F8E9-  F0 08      BEQ  $F8F3        ; ist Subdirectory
F8EB-  C9 40      CMP  #$40
F8ED-  90 04      BCC  $F8F3        ; oder Type 0/1/2/3
F8EF-  A9 4A      LDA  #$4A         ; ERR#4A: INCOMPATIBLE FILE FORMAT
F8F1-  D0 ED      BNE  $F8E0        ; „always"
F8F3-  20 4B F9   JSR  $F94B        ; setzt Pointer auf neuen Path, Copy & Verify
F8F6-  B0 C1      BCS  $F8B9        ; „never" – bereits verifiziert ($F872)
F8F8-  AC B9 FE   LDY  $FEB9        ; Start-Index des unterschiedlichen Namens
F8FB-  BE 00 D7   LDX  $D700,Y
F8FE-  98         TYA
F8FF-  79 00 D7   ADC  $D700,Y      ; Startindex für Kopier-Loop von hinten
F902-  A8         TAY
F903-  B9 00 D7   LDA  $D700,Y      ; der neue Name wird in das File Info
F906-  9D 5F FE   STA  $FE5F,X        kopiert
F909-  88         DEY
F90A-  CA         DEX
F90B-  D0 F6      BNE  $F903
F90D-  AD 5F FE   LDA  $FE5F        ; Storage Type
F910-  29 F0      AND  #$F0
F912-  AA         TAX
F913-  19 00 D7   ORA  $D700,Y      ; + Länge des neuen Namens
F916-  8D 5F FE   STA  $FE5F
F919-  E0 D0      CPX  #$D0
F91B-  D0 1C      BNE  $F939        ; keine Subdirectory
F91D-  AD 70 FE   LDA  $FE70        ; bei Subdirs muß auch der Name im Key
F920-  85 46      STA  $46            Block geändert werden: die Blocknummer
F922-  AD 71 FE   LDA  $FE71          „erster Block" aus dem File Info
F925-  85 47      STA  $47            wird gesetzt
F927-  20 EE EB   JSR  $EBEE        ; und der Key Block gelesen
F92A-  B0 8D      BCS  $F8B9
F92C-  AC B9 FE   LDY  $FEB9
F92F-  B9 00 D7   LDA  $D700,Y
F932-  09 E0      ORA  #$E0         ; => Storage Type im Header
F934-  20 3C F9   JSR  $F93C        ; Kopie des Namens, Rewrite des Blocks
F937-  B0 80      BCS  $F8B9
```

```
F939-   4C C6 E4    JMP   $E4C6          ; ENTRY & UPDATE ohne System Time
```

– Kopiert einen neuen Namen in den Directory Space und schreibt den Block zurück

Xref: $F84B – RENAME Volumes: neuer Volume-Name im Key Block
$F934 – RENAME Subdirectories: neuer Name im Key Block

```
F93C-   A2 00       LDX   #$00          ; Aufruf mit Storage Type & Namenslänge
F93E-   9D 04 DC    STA   $DC04,X
F941-   E8          INX
F942-   C8          INY
F943-   B9 00 D7    LDA   $D700,Y
F946-   D0 F6       BNE   $F93E         ; Loop bis Namensende
F948-   4C EA EB    JMP   $EBEA         ; Rewrite des Blocks
```

– Setzt den Pointer zum neuen Namen und ruft Copy & Verify Pathname auf

Xref: $F82A – RENAME Volumes
$F872 – RENAME Subdirs/Files: Prüfung des neuen Namens + Kopie für Vergleich
$F8F3 – RENAME Subdirs/Files: Holen des neuen Namens für Kopie in File Info

```
F94B-   A0 03       LDY   #$03          ; Parameter $03/04 aus dem „user"-Block
F94D-   B1 40       LDA   ($40),Y       ; Startadresse des neuen Pathnamens
F94F-   C8          INY
F950-   85 48       STA   $48           ; Pointer zum Namen für Copy & Verify
F952-   B1 40       LDA   ($40),Y
F954-   85 49       STA   $49
F956-   4C 95 E0    JMP   $E095         ; Einsprung in Copy & Verify mit
                                          Pointer gesetzt
```

– Lädt den Startindex des momentanen Pathname

Xref: $F863 – RENAME Subdirs/Files: Vor der Kopie des neuen Namens für Vergleich
$F877 – RENAME Subdirs/Files: Reload des Startindex vor endgültiger Kopie

```
F959-   A0 00       LDY   #$00
F95B-   2C BC FE    BIT   $FEBC         ; FULL NAME Flag
F95E-   30 03       BMI   $F963
F960-   AC 9A BF    LDY   $BF9A         ; Pathname nicht vollständig, Start mit
F963-   A2 00       LDX   #$00            System PREFIX
F965-   60          RTS
```

**$C1 – DESTROY COMMAND**; Vorher Copy & Verify Pathname

```
F966-   20 A3 E5    JSR   $E5A3         ; GET FILE
F969-   B0 4A       BCS   $F9B5         ; nicht gefunden oder VolDIR: RTS mit ERR
F96B-   20 B3 EF    JSR   $EFB3         ; sucht den FCB ab: ist dieser File OPEN?
F96E-   AD 97 FE    LDA   $FE97         ; „File gefunden"-Flag, gesetzt von $EFB3
F971-   F0 04       BEQ   $F977
F973-   A9 50       LDA   #$50          ; ERR#50: FILE IS OPEN
F975-   38          SEC
F976-   60          RTS
```

```
F977-  A9 00      LDA  #$00        ; Setzen von BLOCKS NEEDED = 00 00
F979-  8D 94 FE   STA  $FE94
F97C-  8D 95 FE   STA  $FE95

F97F-  20 73 E9   JSR  $E973       ; SPACE ON VOL liefert BLOCKS FREE
F982-  90 05      BCC  $F989
F984-  C9 48      CMP  #$48        ; ERR#48: VOLUME FULL wird maskiert
F986-  38         SEC
F987-  D0 2C      BNE  $F9B5       ; andere Fehler nicht

F989-  AD 7D FE   LDA  $FE7D       ; ACCESS aus dem File Info

F98C-  29 80      AND  #$80        ; DESTROY erlaubt?
F98E-  D0 05      BNE  $F995
F990-  A9 4E      LDA  #$4E        ; ERR#4E: FILE ACCESS ERROR
F992-  20 09 BF   JSR  $BF09       ; SET SYSERR , RTS zu $DE78
F995-  AD 30 BF   LDA  $BF30       ; Unitnummer, gesetzt von GET FILE
F998-  20 65 F4   JSR  $F465       ; Test auf „READY" und WRITE PROTECTED

F99B-  B0 18      BCS  $F9B5
F99D-  AD 70 FE   LDA  $FE70       ; Nummer des „ersten Blocks im File"
F9A0-  8D BF FE   STA  $FEBF         wird von DESTROY stufenweise verändert,
F9A3-  AD 71 FE   LDA  $FE71
F9A6-  8D C0 FE   STA  $FEC0
F9A9-  AD 5F FE   LDA  $FE5F       ; genauso wie der Storage Type.
F9AC-  29 F0      AND  #$F0
F9AE-  C9 40      CMP  #$40        ; Datenfile (Type 1/2/3)?
F9B0-  90 04      BCC  $F9B6
F9B2-  4C 0E FA   JMP  $FA0E       ; nein, muß Subdirectory sein
F9B5-  60         RTS
```

– DESTROY für Datenfiles

```
F9B6-  8D C1 FE   STA  $FEC1       ; Storage Type
F9B9-  A2 05      LDX  #$05
F9BB-  A9 00      LDA  #$00        ; $FEC2/C3: Zähler der entfernten Blocks
F9BD-  9D C1 FE   STA  $FEC1,X       $FEC4: Start mit ersten Indexblock
F9C0-  CA         DEX                $FEC5: Start mit ersten Datenblock
F9C1-  D0 FA      BNE  $F9BD         $FEC6/C7: Start innerhalb des Datenblocks
F9C3-  A9 02      LDA  #$02        ; ist hier $200: kein Löschen im Speicher,
F9C5-  8D C7 FE   STA  $FEC7         weil der Block vollständig entfernt wird.
F9C8-  20 78 FA   JSR  $FA78       ; REMOVE BLOCKS
F9CB-  B0 E8      BCS  $F9B5
```

Xref: $F9CB – DESTROY für Datenfiles
JP $FA47 – DESTROY für Subdirectories

```
F9CD-  AE C0 FE   LDX  $FEC0       ; „erster Block im File" wird jetzt
F9D0-  AD BF FE   LDA  $FEBF         ebenfalls entfernt:
F9D3-  20 34 EA   JSR  $EA34       ; DEALLOCATE BLOCK
F9D6-  B0 DD      BCS  $F9B5
F9D8-  A9 00      LDA  #$00
F9DA-  8D 5F FE   STA  $FE5F       ; Storage Type auf 0 (deleted)
F9DD-  CD 53 FE   CMP  $FE53       ; Anzahl der aktiven Einträge in der DIR:
F9E0-  D0 03      BNE  $F9E5       ; wenn lobyte = $00,
F9E2-  CE 54 FE   DEC  $FE54       ; DEC Hibyte
F9E5-  CE 53 FE   DEC  $FE53
F9E8-  20 93 EB   JSR  $EB93       ; schreibt die VBM zurück
F9EB-  B0 C8      BCS  $F9B5
F9ED-  20 F3 F9   JSR  $F9F3       ; erhöht BLOCKS FREE im VCB entsprechend
F9F0-  4C B6 E4   JMP  $E4B6       ; ENTRY & UPDATE mit Storage Type 0
```

– **BLOCKS FREE** im VCB-Eintrag wird um die Anzahl der entfernten Blocks erhöht

Xref: $F72A – SET EOF nach Verkürzung eines Files
$F9ED – DESTROY

```
F9F3-   AC 91 FE    LDY   $FE91        ; VCB-Index
F9F6-   AD C2 FE    LDA   $FEC2        ; Anzahl der entfernten Blocks - lobyte
F9F9-   79 14 D9    ADC   $D914,Y      ; BLOCKS FREE lo
F9FC-   99 14 D9    STA   $D914,Y
F9FF-   AD C3 FE    LDA   $FEC3        ; Hibytes
FA02-   79 15 D9    ADC   $D915,Y
FA05-   99 15 D9    STA   $D915,Y
FA08-   A9 00       LDA   #$00
FA0A-   99 1C D9    STA   $D91C,Y      ; VBM-Extent 00 (erster VBM-Block)
FA0D-   60          RTS
```

– DESTROY für Storage Types <> 1/2/3

```
FA0E-   C9 D0       CMP   #$D0         ; Storage Type: Subdirectory?
FA10-   D0 4F       BNE   $FA61        ; nein, INCOMPATIBLE FILE FORMAT
FA12-   20 64 EB    JSR   $EB64        ; GET VBM
FA15-   B0 49       BCS   $FA60
FA17-   AD 70 FE    LDA   $FE70        ; die Nummer des „ersten Blocks im File"
FA1A-   85 46       STA   $46            ist hier die Nummer des Key Blocks
FA1C-   AD 71 FE    LDA   $FE71          der Subdirectory (aus dem File Info)
FA1F-   85 47       STA   $47
FA21-   20 EE EB    JSR   $EBEE        ; liest den Key Block auf $DC00
FA24-   B0 3A       BCS   $FA60
FA26-   AD 25 DC    LDA   $DC25        ; Anzahl der aktiven Einträge
FA29-   D0 05       BNE   $FA30          muß 00 00 sein, sonst Fehler
FA2B-   AD 26 DC    LDA   $DC26
FA2E-   F0 05       BEQ   $FA35
FA30-   A9 4E       LDA   #$4E         ; ERR=4E: FILE ACCESS ERROR
FA32-   20 09 BF    JSR   $BF09        ; SET SYSERR, RTS zu $DE78

FA35-   8D 04 DC    STA   $DC04        ; Storage Type/Namenslänge = 00
FA38-   20 EA EB    JSR   $EBEA        ; Rewrite des Blocks
FA3B-   B0 23       BCS   $FA60

FA3D-   AD 02 DC    LDA   $DC02        ; Forward Reference zum nächsten DIR-Block
FA40-   CD 03 DC    CMP   $DC03
FA43-   D0 04       BNE   $FA49        ; wenn 00 00, ist die Directory zu Ende:
FA45-   C9 00       CMP   #$00           der „erste Block" (Key Block) wird bei
FA47-   F0 84       BEQ   $F9CD          $F9CD entfernt

FA49-   AE 03 DC    LDX   $DC03
FA4C-   20 34 EA    JSR   $EA34        ; DEALLOCATE BLOCK
FA4F-   B0 0F       BCS   $FA60
FA51-   AD 02 DC    LDA   $DC02        ; die Nummer des nächsten Blocks wird
FA54-   85 46       STA   $46            gesetzt. Dieser Block ist soeben entfernt
FA56-   AD 03 DC    LDA   $DC03          worden, d.h. der „erste Block im File"
FA59-   85 47       STA   $47            ist nach wie vor in der VBM belegt!
FA5B-   20 EE EB    JSR   $EBEE        ; liest den Block auf $DC00
FA5E-   90 DD       BCC   $FA3D
FA60-   60          RTS                ; RTS nur für Fehler

FA61-   A9 4A       LDA   #$4A         ; ERR=4A: INCOMPATIBLE FILE FORMAT
FA63-   20 09 BF    JSR   $BF09        ; SET SYSERR, RTS zu $DE78
```

– Setzt „FILE ALTERED" („FLUSH always")

Xref: $F394 – JP vom Exit des WRITE Transfer-Loops
$F66A – JP vom Setzen eines neuen EOF im FCB-Eintrag (SET EOF)

```
FA66-   48          PHA
FA67-   98          TYA
FA68-   48          PHA
FA69-   AC 92 FE    LDY   $FE92          ; FCB-Index
FA6C-   B9 1C D8    LDA   $D81C,Y        ; Byte $1C: „File altered"-Flag
FA6F-   09 80       ORA   #$80             für FLUSH („FLUSH always")
FA71-   99 1C D8    STA   $D81C,Y
FA74-   68          PLA
FA75-   A8          TAY
FA76-   68          PLA
FA77-   60          RTS
```

## – REMOVE BLOCKS

Xref: $F6F9 – SET EOF: $FEC4..$FEC6 auf neuen EOF gesetzt
$F9C8 – DESTROY: $FEC4..$FEC6 auf 00 02 00 gesetzt (nur erster Block bleibt)

```
FA78-   AD C1 FE    LDA   $FEC1          ; Storage Type
FA7B-   C9 20       CMP   #$20
FA7D-   90 0B       BCC   $FA8A
FA7F-   C9 30       CMP   #$30
FA81-   90 0A       BCC   $FA8D
FA83-   C9 40       CMP   #$40
FA85-   90 09       BCC   $FA90
FA87-   20 0C BF    JSR   $BF0C          ; <> $10/$20/$30: SYSTEM DEATH
FA8A-   4C 5C FB    JMP   $FB5C          ; REMOVE BLOCKS für Storage Type 1
FA8D-   4C 23 FB    JMP   $FB23          ; REMOVE BLOCKS für Storage Type 2
```

– REMOVE BLOCKS für Storage Type 3

```
FA90-   A9 80       LDA   #$80           ; Start vom höchstmöglichen Block her
FA92-   8D C8 FE    STA   $FEC8          ; Index im MASTER
FA95-   20 87 FB    JSR   $FB87          ; liest „ersten Block im File" – hier MASTER
FA98-   B0 5B       BCS   $FAF5
FA9A-   AC C8 FE    LDY   $FEC8
FA9D-   CC C4 FE    CPY   $FEC4          ; UNTERGRENZE IM MASTER-BLOCK
FAA0-   F0 54       BEQ   $FAF6          ; ist erreicht => REMOVE BLOCKS Type 2

FAA2-   A2 07       LDX   #$07           ; es werden bis zu 8 existierende (<> 00 00)
FAA4-   B9 00 DC    LDA   $DC00,Y          Indexblocknummern auf $FECA...D2 kopiert.
FAA7-   9D CA FE    STA   $FECA,X          Dieser Loop startet mit $80, d.h. bei
FAAA-   19 00 DD    ORA   $DD00,Y          einem File < 16 MByte werden erst einmal
FAAD-   F0 09       BEQ   $FAB8            alle Indexblocknummern, die nie belegt
FAAF-   B9 00 DD    LDA   $DD00,Y          waren, übersprungen.
FAB2-   9D D2 FE    STA   $FED2,X
FAB5-   CA          DEX
FAB6-   30 12       BMI   $FACA
FAB8-   88          DEY
FAB9-   CC C4 FE    CPY   $FEC4          ; Untergrenze erreicht?
FABC-   D0 E6       BNE   $FAA4
FABE-   C8          INY
FABF-   A9 00       LDA   #$00           ; ja, Auffüllen der restlichen Tabelle mit
FAC1-   9D CA FE    STA   $FECA,X          Nullen (= Tabellenende)
FAC4-   9D D2 FE    STA   $FED2,X
FAC7-   CA          DEX
FAC8-   10 F7       BPL   $FAC1
```

– Loop durch diese (maximal) 8 kopierten Indexblocknummern

```
FACA-   88          DEY
FACB-   8C C8 FE    STY   $FEC8        ; Index für nächste Kopie von 8 Indexblocks
FACE-   A2 07       LDX   #$07
FAD0-   8E C9 FE    STX   $FEC9        ; Herunterzähler
FAD3-   BD CA FE    LDA   $FECA,X
FAD6-   85 46       STA   $46          ; Setzen der Blocknummer
FAD8-   1D D2 FE    ORA   $FED2,X        und Test auf Tabellenende
FADB-   F0 B8       BEQ   $FA95        ; Tabellenende erreicht
FADD-   BD D2 FE    LDA   $FED2,X
FAE0-   85 47       STA   $47
FAE2-   20 EE EB    JSR   $EBEE        ; liest Indexblock auf $DC00: der MASTER-
FAE5-   B0 0E       BCS   $FAF5          Block wird dabei im Speicher überschrieben
FAE7-   20 B6 FB    JSR   $FBB6        ; entfernt alle Datenblocks dieser  Indizes,
FAEA-   B0 09       BCS   $FAF5          die Indexblocks selber bleiben belegt!
FAEC-   AE C9 FE    LDX   $FEC9        ; Herunterzähler
FAEF-   CA          DEX
FAF0-   10 DE       BPL   $FAD0        ; Loop
FAF2-   30 A1       BMI   $FA95        ; nächste 8 Indexblocks

FAF4-   18          CLC                ; kein Exit für Type 3 - Loop!
FAF5-   60          RTS
```

– Entfernung der Indexblocks aus dem MASTER-Block

Wenn keine weiteren ganzen Indexblocks mehr zu entfernen sind, wird der MASTER-Block ein weiteres Mal geladen ($FA95) und hierher gesprungen ($FA8D).

```
FAF6-   AC C4 FE    LDY   $FEC4        ; Untergrenze im MASTER
FAF9-   C8          INY                ; + 1, letzter Indexblock bleibt!
FAFA-   20 B8 FB    JSR   $FBB8        ; entfernt die Indexblocks selber ab der
FAFD-   B0 F6       BCS   $FAF5          Untergrenze aus VBM und MASTER-Block
FAFF-   20 EA EB    JSR   $EBEA        ; Rewrite des MASTER-Blocks
FB02-   B0 F1       BCS   $FAF5
FB04-   AC C4 FE    LDY   $FEC4        ; Sprung, wenn Untergrenze = 00, d.h.
FB07-   F0 15       BEQ   $FB1E          alle Indexblocks entfernt
FB09-   B9 00 DC    LDA   $DC00,Y      ; sonst wird der jetzt letzte Indexblock
FB0C-   85 46       STA   $46            gesetzt und geladen
FB0E-   19 00 DD    ORA   $DD00,Y
FB11-   F0 E1       BEQ   $FAF4        ; „sparse" Indexblock: CLC/RTS
FB13-   B9 00 DD    LDA   $DD00,Y
FB16-   85 47       STA   $47
FB18-   20 EE EB    JSR   $EBEE        ; liest den jetzt letzten Indexblock
FB1B-   90 0B       BCC   $FB28
FB1D-   60          RTS

FB1E-   20 94 FB    JSR   $FB94        ; MASTER-Block ist leer: DEALLOCATE des
FB21-   B0 D2       BCS   $FAF5          MASTER-Blocks selber, Type-Herabsetzung
```

– REMOVE BLOCKS für Storage Type 2

Xref: $FA8D – JP – File hat Storage Type 2
$FB1E – MASTER Block entfernt, File hat jetzt Type 2

```
FB23-   20 87 FB    JSR   $FB87        ; liest den „ersten Block im File"
FB26-   B0 CD       BCS   $FAF5
```

Xref: $FB1B – SET EOF: „erster Block" ist immer noch MASTER, letzter Index geladen

```
FB28-   AC C5 FE    LDY   $FEC5        ; UNTERGRENZE IM INDEXBLOCK
FB2B-   C8          INY                ; letzter Datenblock bleibt bestehen!
```

```
FB2C-   F0 0A       BEQ   $FB38        ; Sprung bei Untergrenze = $FF, kein REMOVE
FB2E-   20 B8 FB    JSR   $FBB8        ; entfernt alle Datenblocks ab Untergrenze
FB31-   B0 C2       BCS   $FAF5          aus dem Indexblock (bzw. aus der VBM)
FB33-   20 EA EB    JSR   $EBEA        ; Rewrite des Indexblocks
FB36-   B0 BD       BCS   $FAF5
FB38-   AC C5 FE    LDY   $FEC5        ; war Untergrenze = 00, d.h. Inhalt
FB3B-   F0 15       BEQ   $FB52          komplett entfernt?
FB3D-   B9 00 DC    LDA   $DC00,Y      ; nein, Setzen des jetzt letzten Datenblocks
FB40-   85 46       STA   $46
FB42-   19 00 DD    ORA   $DD00,Y
FB45-   F0 AD       BEQ   $FAF4        ; „sparse" Datenblock: CLC/RTS
FB47-   B9 00 DD    LDA   $DD00,Y
FB4A-   85 47       STA   $47
FB4C-   20 EE EB    JSR   $EBEE        ; liest den Datenblock auf $DC00..
FB4F-   90 10       BCC   $FB61
FB51-   60          RTS
```

– Inhalt des Indexblocks komplett entfernt:

```
FB52-   AD C4 FE    LDA   $FEC4        ; Untergrenze im MASTER ebenfalls 00 ?
FB55-   D0 E6       BNE   $FB3D        ; nein, File hat Type 3!
FB57-   20 94 FB    JSR   $FB94        ; ja, Entfernung des Indexblocks aus der
FB5A-   B0 2A       BCS   $FB86          VBM und Herabsetzen des Storage Type
```

– REMOVE BLOCKS für Storage Type 1

Xref: $FA8A – JP: File hat Storage Type 1
$FB59 – File hat jetzt Storage Type 1 – einziger Indexblock komplett entfernt

```
FB5C-   20 87 FB    JSR   $FB87        ; liest „ersten Block im File"
FB5F-   B0 25       BCS   $FB86
```

Xref: $FB4F – File hat immer noch einen höheren Type – letzter Datenblock geladen

```
FB61-   AC C7 FE    LDY   $FEC7        ; $FEC6/C7: Untergrenze für Löschung
FB64-   F0 06       BEQ   $FB6C          innerhalb dieses Datenblocks. Für
FB66-   88          DEY                  DESTROY auf $200 gesetzt, d.h. keine
FB67-   D0 1C       BNE   $FB85          Löschung, der Datenblock wird komplett
FB69-   AC C6 FE    LDY   $FEC6          entfernt. Für SET EOF beliebige Werte
FB6C-   A9 00       LDA   #$00           im Bereich 00 00 bis 01 FF
FB6E-   99 00 DD    STA   $DD00,Y
FB71-   C8          INY
FB72-   D0 FA       BNE   $FB6E
FB74-   AC C7 FE    LDY   $FEC7
FB77-   D0 09       BNE   $FB82
FB79-   AC C6 FE    LDY   $FEC6        ; wenn $FEC6/C7 einen Wert <> $200 haben,
FB7C-   99 00 DC    STA   $DC00,Y        wird der Datenblock nach Löschung
FB7F-   C8          INY                  im Speicher zurückgeschrieben
FB80-   D0 FA       BNE   $FB7C
FB82-   4C EA EB    JMP   $EBEA        ; schreibt den Block zurück
FB85-   18          CLC                ; ansonsten nur RTS
FB86-   60          RTS
```

– Liest den „ersten Block im File" auf $DC00/$DD00

Xref: $FA95 – Lesen des MASTER-Blocks
$FB23 – Lesen des (einzigen) Indexblocks
$FB5C – Lesen des (einzigen) Datenblocks

```
FB87-   AD BF FE    LDA   $FEBF        ; Nummer des „ersten Blocks im File",
FB8A-   85 46       STA   $46            von DESTROY/SET EOF aus dem File Info
FB8C-   AD C0 FE    LDA   $FEC0          kopiert und bei Type-Herabsetzung
FB8F-   85 47       STA   $47            verändert
FB91-   4C EE EB    JMP   $EBEE        ; RTS von da aus
```

– Entfernt den „ersten Block" und erniedrigt den Storage Type des Files

Xref: $FB1E – Entfernung des MASTER, „erster Block" ist danach (einziger) Indexblock
$FB57 – Entfernung des einzigen Indexblocks, „erster Block" danach Datenblock

```
FB94-   AE C0 FE    LDX   $FEC0        ; Nummer des „ersten Blocks im File"
FB97-   AD BF FE    LDA   $FEBF        ; Lobyte
FB9A-   20 34 EA    JSR   $EA34        ; DEALLOCATE BLOCK
FB9D-   B0 16       BCS   $FBB5
FB9F-   AD 00 DC    LDA   $DC00        ; enthält die Blocknummer des jeweils
FBA2-   8D BF FE    STA   $FEBF          „nächstniedrigeren" Block-Typen und
FBA5-   AD 00 DD    LDA   $DD00          ist als einzige Blockno bis jetzt
FBA8-   8D C0 FE    STA   $FEC0          nicht entfernt worden. Wird als neuer
FBAB-   38          SEC                  „erster Block" gesetzt.
FBAC-   AD C1 FE    LDA   $FEC1
FBAF-   E9 10       SBC   #$10         ; Storage Type – $10, d.h. Verminderung
FBB1-   8D C1 FE    STA   $FEC1          um 1
FBB4-   18          CLC
FBB5-   60          RTS
```

– Entfernung von Blocks, auf die ein MASTER/Indexblock zeigt

Xref: $FAE7 – Entfernung aller Datenblocks aus einem Indexblock

```
FBB6-   A0 00       LDY   #$00
```

Xref: $FAFA – Entfernung von Indexblock aus dem MASTER ab Untergrenze
$FB2E – Entfernung von Datenblocks aus einem Indexblock ab Untergrenze

```
FBB8-   A5 46       LDA   $46
FBBA-   48          PHA
FBBB-   A5 47       LDA   $47          ; die momentane Blocknummer ist die
FBBD-   48          PHA                  des MASTER/Indexblocks, speichern
FBBE-   8C 9D FE    STY   $FE9D        ; Untergrenze
FBC1-   B9 00 DC    LDA   $DC00,Y
FBC4-   D9 00 DD    CMP   $DD00,Y
FBC7-   D0 04       BNE   $FBCD
FBC9-   C9 00       CMP   #$00
FBCB-   F0 13       BEQ   $FBE0        ; Blockno = 00 00, kein DEALLOCATE nötig
FBCD-   BE 00 DD    LDX   $DD00,Y
FBD0-   20 34 EA    JSR   $EA34        ; DEALLOCATE BLOCK – liest/schreibt evtl.
FBD3-   B0 0F       BCS   $FBE4          VBM-Blocks (Extents)
FBD5-   AC 9D FE    LDY   $FE9D
FBD8-   A9 00       LDA   #$00
```

```
FBDA-   99 00 DC    STA   $DC00,Y       ; Block aus der VBM entfernt - jetzt
FBDD-   99 00 DD    STA   $DD00,Y         Löschen des Eintrags im MASTER/Index
FBE0-   C8          INY
FBE1-   D0 DB       BNE   $FBBE         ; Loop bis Blockende
FBE3-   18          CLC
FBE4-   AA          TAX                 ; 00 oder ERR# im Fehlerfall
FBE5-   68          PLA
FBE6-   85 47       STA   $47           ; Restore der Blocknummer
FBE8-   68          PLA
FBE9-   85 46       STA   $46
FBEB-   8A          TXA
FBEC-   60          RTS                 ; mit A <> 00 im Fehlerfall, kein Carry!
```

– Setzt einen Filepuffer, Startadresse aus dem „user"-Parmblock

Xref: $EF54 – OPEN: Startadresse des Filepuffers sind Bytes $04/05 im Parmblock

```
FBED-   A0 04       LDY   #$04
```

Xref: $FCBF – SET BUFFER: Neuer Pufferstart sind Bytes $03/04 im Parmblock

```
FBEF-   B1 40       LDA   ($40),Y       ; Pufferstartadresse hi
FBF1-   AA          TAX
FBF2-   C9 08       CMP   #$08          ; muß im Bereich $0800-$BBFF liegen
FBF4-   90 42       BCC   $FC38
FBF6-   C9 BC       CMP   #$BC
FBF8-   B0 3E       BCS   $FC38
FBFA-   85 4B       STA   $4B
FBFC-   88          DEY
FBFD-   B1 40       LDA   ($40),Y
FBFF-   85 4A       STA   $4A           ; $004A/4B: Puffer-Startadresse
FC01-   D0 35       BNE   $FC38
FC03-   E8          INX
FC04-   E8          INX
FC05-   E8          INX
FC06-   E8          INX                 ; die Pufferlänge ist $400 Bytes, d.h.
FC07-   CA          DEX                   die Endadresse ist 3 Speicherseiten höher
FC08-   20 73 FC    JSR   $FC73         ; berechnet Y-Index und Bit Mask
FC0B-   39 58 BF    AND   $BF58,Y       ; System Bit Map
FC0E-   D0 28       BNE   $FC38         ; Speicherseite nicht frei
FC10-   E4 4B       CPX   $4B           ; Loop abwärts für alle vier Seiten
FC12-   D0 F3       BNE   $FC07
FC14-   E8          INX
FC15-   E8          INX
FC16-   E8          INX
FC17-   E8          INX
FC18-   CA          DEX
FC19-   20 73 FC    JSR   $FC73         ; derselbe Loop noch einmal, diesmal wird
FC1C-   19 58 BF    ORA   $BF58,Y         das entsprechende Bit in der System
FC1F-   99 58 BF    STA   $BF58,Y         Bit Map belegt
FC22-   E4 4B       CPX   $4B
FC24-   D0 F2       BNE   $FC18
FC26-   AC 92 FE    LDY   $FE92         ; FCB-Index für das File
FC29-   B9 00 D8    LDA   $D800,Y       ; Reference Nummer
FC2C-   0A          ASL
FC2D-   99 0B D8    STA   $D80B,Y       ; * 2 ist Index in der BUFFER TABLE
FC30-   AA          TAX
FC31-   A5 4B       LDA   $4B
FC33-   9D 6F BF    STA   $BF6F,X       ; Pufferstartadresse => System Global Page
```

```
FC36-   18          CLC
FC37-   60          RTS

FC38-   A9 56       LDA   #$56          ; ERR#56: BAD BUFFER ADDRESS
FC3A-   38          SEC
FC3B-   60          RTS
```

– Kopiert eine Puffer-Startadresse aus der BUFFER TABLE (SGP) auf $FEDD/DE

Xref: $E1EE – SET FILE DATA
$FC4A – DEALLOCATE FILEBUFFER: OPEN Error Exit und CLOSE

```
FC3C-   AA          TAX                 ; Aufruf mit File Refno * 2 im Akku
FC3D-   BD 6E BF    LDA   $BF6E,X       ; BUFFER TABLE – Lobyte
FC40-   8D DD FE    STA   $FEDD
FC43-   BD 6F BF    LDA   $BF6F,X       ; Hibytes
FC46-   8D DE FE    STA   $FEDE
FC49-   60          RTS
```

– **DEALLOCATE FILEBUFFER** (aus der System Bit Map und aus der BUFFER TABLE)

Xref: $EF86 – OPEN Error Exit
$F4C7- CLOSE

```
FC4A-   20 3C FC    JSR   $FC3C         ; kopiert die Puffer-Startadresse
FC4D-   09 00       ORA   #$00
FC4F-   F0 20       BEQ   $FC71         ; kein Puffer existent: CLC/RTS
FC51-   A9 00       LDA   #$00
FC53-   9D 6F BF    STA   $BF6F,X       ; sonst wird die Startadresse in der
FC56-   9D 6E BF    STA   $BF6E,X         BUFFER TABLE gelöscht
```

Xref: $FCCE – SET BUF: Startadresse in der BUFFER TABLE enthält bereits neuen Puffer

```
FC59-   AE DE FE    LDX   $FEDE         ; Puffer-Startadresse hi
FC5C-   E8          INX
FC5D-   E8          INX
FC5E-   E8          INX
FC5F-   E8          INX                 ; Pufferlänge ist $400 Byte, d.h. das Ende
FC60-   CA          DEX                   des Puffers liegt 3 Speicherseiten höher
FC61-   20 73 FC    JSR   $FC73         ; berechnet Bit Mask und Y-Index
FC64-   49 FF       EOR   #$FF          ; Complement
FC66-   39 58 BF    AND   $BF58,Y       ; das entsprechende Bit in der System
FC69-   99 58 BF    STA   $BF58,Y         BIT MAP wird wieder gelöscht, d.h.
FC6C-   EC DE FE    CPX   $FEDE           die Speicherseite wieder freigegeben
FC6F-   D0 EF       BNE   $FC60         ; Loop abwärts bis zum Pufferstart
FC71-   18          CLC
FC72-   60          RTS
```

– Berechnet den Y-Index und die Bit Mask für eine Speicherseite in der SBM

Xref: $FC08 – Setzen des Puffers mit „user"-Parametern: Check mit System Bit Map
$FC19 – Setzen des Puffers mit „user"-Parametern: Belegung in SBM
$FC61 – DEALLOCATE BUFFER
$FCA0 – Test eines beliebigen Bereichs gegen SBM

```
FC73-   8A          TXA                 ; Aufruf mit Hibyte des Adresse in X-Reg
FC74-   29 07       AND   #$07
FC76-   A8          TAY
FC77-   B9 00 FE    LDA   $FE00,Y       ; Bit Mask: $80/$40/$20/$10/$08/$04/$02/$01
FC7A-   48          PHA
FC7B-   8A          TXA
FC7C-   4A          LSR                 ; Speicherseiten-Nummer durch 8 ist Index
FC7D-   4A          LSR                   in der SBM - jedes SBM-Byte enthält
FC7E-   4A          LSR                   8 Speicherseiten
FC7F-   A8          TAY
FC80-   68          PLA                 ; Bit Mask
FC81-   60          RTS
```

**– Check eines beliebigen Speicherbereichs gegen die System Bit Map**

Xref: $E1A3 – GET PREFIX: Test des Zielpuffers für das zurückerhaltene PREFIX
$E252 – ONLINE: Test des Zielpuffers für die Volume-Namen
$F046 – READ: Test des Zielbereichs

```
FC82-   A5 4F       LDA   $4F           ; Aufruf mit $004E/4F = Startadresse des
FC84-   C9 02       CMP   #$02            Zielpuffers, muß > $200 sein
FC86-   90 B0       BCC   $FC38
FC88-   AE DB FE    LDX   $FEDB         ; Pufferlänge in $FEDA/DB
FC8B-   AD DA FE    LDA   $FEDA         ; wenn die Länge = $xx00 oder $xx01 ist,
FC8E-   E9 01       SBC   #$01            wird eine Speicherseite weniger
FC90-   B0 01       BCS   $FC93           benötigt
FC92-   CA          DEX
FC93-   18          CLC
FC94-   65 4E       ADC   $4E
FC96-   8A          TXA
FC97-   65 4F       ADC   $4F
FC99-   AA          TAX
FC9A-   E0 BF       CPX   #$BF          ; Pufferende größer $BEFF?
FC9C-   B0 9A       BCS   $FC38         ; ja, BAD BUFFER ADDRESS
```

Xref: $DEC9 – BLOCKREAD/WRITE: $004E/4F und X (Herunterzähler von Obergrenze) bereits gesetzt, kein Test auf > $BFFF (s.„Spezielles")

```
FC9E-   E8          INX
FC9F-   CA          DEX                 ; berechnet Y-Index und Bit Mask
FCA0-   20 73 FC    JSR   $FC73         ; System Bit Map
FCA3-   39 58 BF    AND   $BF58,Y       ; Seite belegt, BAD BUFFER ADDRESS
FCA6-   D0 90       BNE   $FC38
FCA8-   E4 4F       CPX   $4F           ; Loop abwärts, bis die Startadresse
FCAA-   D0 F3       BNE   $FC9F           wieder erreicht ist
FCAC-   18          CLC
FCAD-   60          RTS
```

**$D3 – GET BUFFER COMMAND** ; File muß OPEN sein – Set File Data vorher

```
FCAE-   A0 02       LDY   #$02
FCB0-   AD DD FE    LDA   $FEDD         ; $FEDD/DE: Startadresse des Filepuffers,
FCB3-   91 40       STA   ($40),Y         gesetzt durch Set File Data,
FCB5-   C8          INY
FCB6-   AD DE FE    LDA   $FEDE         ; wird in den Parameterblock des
FCB9-   91 40       STA   ($40),Y         „user"-Programms kopiert.
FCBB-   18          CLC
FCBC-   60          RTS                 ; => $DE78
```

**$D2 – SET BUFFER COMMAND** ; File muß OPEN sein – Set File Data vorher

```
FCBD-   A0 03       LDY   #$03
FCBF-   20 EF FB    JSR   $FBEF       ; holt die neue Puffer-Startadresse aus dem
FCC2-   B0 20       BCS   $FCE4         „user"-Parmblock und checkt mit SBM, der
FCC4-   AD DE FE    LDA   $FEDE         neue Puffer wird dabei belegt und in der
FCC7-   85 4F       STA   $4F           BUFFER TABLE eingetragen
FCC9-   AD DD FE    LDA   $FEDD
FCCC-   85 4E       STA   $4E
FCCE-   20 59 FC    JSR   $FC59       ; DEALLOCATE FILEBUFFER - alter Filepuffer,
FCD1-   A0 00       LDY   #$00          2. Einsprung ohne Löschen der Startadresse
FCD3-   A2 03       LDX   #$03          in der BUFFER TABLE
FCD5-   B1 4E       LDA   ($4E),Y
FCD7-   91 4A       STA   ($4A),Y     ; der Pufferinhalt wird in den neuen
FCD9-   C8          INY                 Filepuffer umkopiert
FCDA-   D0 F9       BNE   $FCD5
FCDC-   E6 4B       INC   $4B
FCDE-   E6 4F       INC   $4F
FCE0-   CA          DEX
FCE1-   10 F2       BPL   $FCD5       ; Loop für $400 Byte
FCE3-   18          CLC
FCE4-   60          RTS               ; => $DE78
```

**$65 – REBOOT COMMAND HANDLER** ; keine anderen Routinen vorher

```
FCE5-   AD 83 C0    LDA   $C083       ; BANKSELECT der Bank 2
FCE8-   AD 83 C0    LDA   $C083       ; + WRITE ENABLE (unnötig)
FCEB-   A5 00       LDA   $00
FCED-   48          PHA
FCEE-   A5 01       LDA   $01         ; die Speicherstellen $0000-$0003 werden
FCF0-   48          PHA                 zwischengespeichert
FCF1-   A5 02       LDA   $02
FCF3-   48          PHA
FCF4-   A5 03       LDA   $03
FCF6-   48          PHA
FCF7-   A9 10       LDA   #$10        ; Startadresse: $1000
FCF9-   85 03       STA   $03
FCFB-   A9 D1       LDA   #$D1        ; REBOOT steht in der Bank 2 ab $D100
FCFD-   85 01       STA   $01
FCFF-   A9 00       LDA   #$00
FD01-   85 00       STA   $00
FD03-   85 02       STA   $02
FD05-   A8          TAY
FD06-   A2 03       LDX   #$03        ; REBOOT ist $300 Bytes lang
FD08-   88          DEY
FD09-   B1 00       LDA   ($00),Y
FD0B-   91 02       STA   ($02),Y     ; und wird schlicht umkopiert
FD0D-   98          TYA
FD0E-   D0 F8       BNE   $FD08
FD10-   E6 01       INC   $01
FD12-   E6 03       INC   $03
FD14-   CA          DEX
FD15-   D0 F1       BNE   $FD08       ; Loop für $300 Bytes
FD17-   68          PLA
FD18-   85 03       STA   $03
FD1A-   68          PLA
FD1B-   85 02       STA   $02         ; Restore der Speicherstellen $0000-$0003:
FD1D-   68          PLA                 das ist zwar sehr korrekt, ein Grund
FD1E-   85 01       STA   $01           für diese Zwischenspeicherung ist
FD20-   68          PLA                 allerdings beim momentanen REBOOT
FD21-   85 00       STA   $00          nicht gegeben.
FD23-   AD 8B C0    LDA   $C08B
```

```
FD26-   AD 8B C0    LDA   $C08B        ; BANKSELECT Bank 1 (+ WRITE ENABLE)
FD29-   A9 00       LDA   #$00
FD2B-   8D F2 03    STA   $03F2
FD2E-   A9 10       LDA   #$10         ; der RESET-Vektor wird auf $1000, also
FD30-   8D F3 03    STA   $03F3          auf den Anfang des kopierten REBOOT
FD33-   49 A5       EOR   #$A5           gesetzt.
FD35-   8D F4 03    STA   $03F4
FD38-   4C 00 10    JMP   $1000        ; Exec REBOOT
```

$FD3B..$FD64 Die „Mystery Routine" ist in den Versionen vor 1.1.1 nicht implementiert und im Kapitel „Interrupts der Version 1.1.1" gelistet

## – Tabellen

```
FD65- D3 00 00 00 40 41 00 00  80 81 82 65 C0 C1 C2 C3  ; COMMAND-Bytes
      C4 C5 C6 C7 C8 C9 CA CB  CC CD CE CF 00 D0 D1 D2

FD85- 02 FF FF FF 02 01 FF FF  03 03 00 04 07 01 02 07  ; Anzahl der Parameter
      0A 02 01 01 03 03 04 04  01 01 02 02 FF 02 02 02    für die COMMANDS

FDA5- 02 E3 66 F9 21 F8 EA F7  85 F7 35 E2 35 E1 8D E1  ; Startadressen für die
FDB5- C6 EE 73 F7 01 F0 86 F2  81 F4 F2 F4 2B EC 15 EC    COMMANDS $C0 bis $D3
FDC5- 11 F6 61 F7 BD FC AE FC

FDCD- A0 A1 A2 A3 84 05 06 07  88 49 4A 4B  ; $C0-$D3 CMDS: Bit 0-4 = Index f.$FDA5,
      2C 2D 4E 4F 50 51 52 53                 Bit 7-5 = Flags für „Vorher"-Routinen

FDE1- 01 00 00 02 00    ; Anhängen eines Subdir-Blocks: BLOCKS USED + 1, EOF + $200
```

In einen Subdirectory Key kopiert

```
FDE6- 75 48 55 53 54 4F 4E 21                          ; uHUSTON!
FDEE- 00 00             ; Für Check mit Backward Reference: (00 00 = Key Block)
FDF0- 00 00             ; MLI-Version und „Min.Version"
FDF2- C3 27 0D          ; ACCESS, Eintragslänge, Anzahl Einträge per Block
FDF5- 00 00 00


FDF8- 0F 02 00 04 00 00 08 00   ; FIND FILE: „gefälschtes" File Info für VolDIR:
                                  Type $0F (nicht benutzt): „erster Block" 00 02
                                  BLOCKS USED 00 04, EOF 00 08 00

FDF0- 80 40 20 10 08 04 02 01   ; Bit Mask für System Bit Map / VBM

FDF8- 0C 0D 18 19 15 16 17 1E 10 1F 20  ; Adress-Übersetzung für SET/GET FILE INFO
      80 93 94 21 22 23 24 18 19 1A 1B    MSB gesetzt: Überspringen des Null-Field
```

Text von SYSTEM DEATH, wird direkt in den Bildspeicher kopiert

```
FE1E- A0 C9 CE D3 C5 D2 D4 A0  D3 D9 D3 D4 C5 CD A0 C4      INSERT SYSTEM D
FE2E- C9 D3 CB A0 C1 CE C4 A0  D2 C5 D3 D4 C1 D2 D4 A0     ISK AND RESTART
FE3E- A0 A0 A0 A0 A0 A0 A0 A0

**********************

FE46..FEFF  Variablenbereich

FF00..FF7F  Interface zur RAMDisk-RWTB

FF80..FF9A  Unbenutzt (Nullen)

FF9B..FFFF  IRQ/RESET/NMI Handler, im Kapitel „Interrupts/RESET" besprochen.
```

## 6.2.7. Die MLI-Variablen der Version 1.1.1

Im Gegensatz zu den MLI-Versionen vor 1.1.1 kommt hier der Bereich des Pathname, FCB, VCB, etc. speicherplatzmäßig vor dem MLI und den MLI-Variablen:

$D700..$D740 Pathname, aus dem Parameterblock des „user"-Programms von Copy & Verify Pathname ($E08A) hierher kopiert.

$D741..$D7AB Thunderclock-Routine ($82 – SET TIME Command)

$D7AB..$D7BF Tabellen für die Jahrfeld-Umrechnung der Thunderclock.

$D7C0..$D7FF System Prefix. Das System Prefix endet immer auf $D7FF, die Speicherstelle $BF94 enthält entweder den Startindex im Bereich $C0-$FE oder den Wert $00 (kein Prefix).

---

$D800..$D8FF **FCB** – Im Gegensatz zum VCB werden hier die Einträge in aufsteigender Reihenfolge gemacht. Auch hier umfaßt jeder Eintrag $20 Bytes, d.h. es sind maximal 8 Einträge möglich.

Ein Beispiel für den ersten FCB-Eintrag:

```
D800: 01      Nummer des Eintrags (immer Index/$20) – „Eintrag belegt"
D801: 60      Unitnummer des Files, hier S6, D1
D802: 02 00   Nummer des Key Blocks der Directory, in der der File verzeichnet
              ist (hier 00 02: Volume Directory).
D804: 03 00   Nummer des Directory-Blocks, in dem der Eintrag für den File
              steht (hier der zweite Block der Volume Directory).
D806: 01      Eintragsnummer: Der Eintrag ist der erste innerhalb des Blocks.
D807: 02      Storage Type
D808: E0      STATUS: sowohl Index- als auch ein Datenblock befinden sich im
              Filepuffer und wurden verändert, sie müssen noch zurückgeschrieben
              werden. „File accessed" (Bit 4) ist damit ebenfalls gesetzt.
D809: 03      ACCESS: Nur READ/WRITE werden in den FCB kopiert. Hier ist beides
              erlaubt.
D80A: 00      NEWLINE CHARACTER (hier nicht gesetzt).
D80B: 02      Puffer Index: der File hat den ersten in der Global Page
              verzeichneten Puffer.
D80C: 09 00   „erster Block im File" (00 09)
D80E: 09 00   „Indexblock im Speicher": der (einzige) Indexblock ist geladen
D810: 0F 00   „Datenblock im Speicher" (00 0F).
D812: FD0700  „Position jetzt"(00 07 FD): innerhalb des vierten Datenblocks.
D815: FD0700  EOF (00 07 FD): nach einem WRITE gerade nach hinten verschoben.
D818: 00 05   BLOCKS-USED: der File benötigt 00 05 Blocks
D81A: 00      nicht benutzt
D81B: 07      „LEVEL" – Kopie von der Global Page
D81C: 80      „File altered" – der Eintrag muß bei FLUSH zurückgeschrieben
              werden, sonst 00.
D81D: 00 00   nicht benutzt
D81F: 00      NEWLINE MASK, hier nicht gesetzt.
```

$D820..$D83F Zweiter FCB-Eintrag
$D840..$D85F Dritter FCB-Eintrag
.....
$D8C0..$D8FF Achter FCB-Eintrag

$D900..$D9FF **VCB** – die Einträge werden hier in absteigender Reihenfolge gemacht, d.h. der erste Eintrag, der belegt wird, hat die Basis $D9C0. Jeder Eintrag umfaßt $20 Byte, d.h. es sind maximal 8 Einträge möglich.
Da die Adressierung grundsätzlich über den VCB-Index geschieht, folgt hier ein Beispiel-Eintrag auf $D900:

```
D900: 0x     Länge des folgenden Namens (hier: 4) und „besetzt"-Indikator
D901..0F     NAME in ASCII-Großbuchstaben, z.b. „DEMO"
D910: 60     Nummer des Units, für das dieser Eintrag steht. Hier: S6,D1
D911: 00     „OPEN File(s)": $00 – kein offener File, ansonsten $80
D912: 01 18  Gesamtzahl der Blocks auf dem Volume (DiskII: $0118)
D914: 01 0E  Anzahl der freien Blocks, muß via „Space on Vol" bestimmt werden.
D916: 02 00  Blocknummer des Key Blocks der Volume Directory (00 02)
D918: 00 00  nicht benutzt
D91A: 06 00  Blocknummer des (ersten) VBM-Blocks (00 06)
D91C: 00     VBM-„Extent", hier erster Block
D91D: 00     nicht benutzt
D91E: 00     Anzahl der momentan offenen Files, hier keine
D91F: 00     nicht benutzt.
```

$D920..$D93F nächster („siebter") VCB-Eintrag
$D940..$D95F sechster VCB-Eintrag
....
$D9C0..$D9FF erster VCB-Eintrag

$DA00..$DBFF VBM Space – dieser Speicherplatz wird nur für VBM-Blocks benutzt.

---

$DC00..$DDFF Directory Space – FIND FILE ENTRY lädt Directory-Blocks hierher, REMOVE BLOCKS benutzt diesen Platz auch für Master/Index- und Datenblocks, weil u.U. (DESTROY) kein Filepuffer zur Verfügung steht.

$DE00..$FE45 Das MLI selber, Ende mit Tabellen und Text für SYSTEM DEATH.

Die Variablen und temporären Speicherplätze, die vom MLI benutzt werden, befinden sich im Bereich $FE46..$FEFF. Einige Speicherstellen werden überhaupt nicht benutzt, andere – speziell die „Scratches" (= temporär benutzte Zwischenspeicher) – haben verschiedene Zwecke und sind deshalb nicht eindeutig definierbar. Ihre Bedeutung ist von der Routine abhängig, die diese Speicherstellen gerade benutzt.
Wenn Speicherstellen zu Zweier- und Dreiergruppen zusamengefaßt werden (d.h. für 16- und 24-Bit-Werte), ist die Reihenfolge immer Lo-Hi, d.h. das niederwertigste Byte hat die niedrigere Speicheradresse. Eine Ausnahme gibt es auch hier: die Speicherstellen, die von SET EOF und DESTROY für die Routine REMOVE BLOCKS gesetzt werden ($FEC1-$FEC7) gehen etwas „durcheinander", was die „Wertigkeit" betrifft.

Die RWTB für die DISKII hat (korrekterweise) ihren eigenen Variablenbereich!

Befindet sich der gesuchte Eintrag in einer Subdirectory, werden die folgenden Speicherstellen belegt, ansonsten bekommen sie den Wert $00:

| | |
|---|---|
| $FE46/$FE47 | Blocknummer der Directory, wo die Subdirectory als Eintrag steht („Parent Block"). |
| $FE48 | Eintragsnummer innerhalb dieses Blocks. |
| $FE49 | Länge eines Eintrags innerhalb dieses Blocks. |

Für alle gelesenen Directories, egal ob Volume oder Subdirectory, wird belegt:

| | |
|---|---|
| $FE4A..$4D | CREATION DATE & TIME, bzw. FORMAT DATE & TIME, wenn es sich bei der gelesenen Directory um eine Volume Directory handelt. Diese Speicherstellen werden nirgendwo wieder gelesen. |
| $FE4E/4F | VERSION – diese Speicherstellen werden nirgendwo wieder gelesen. |
| $FE50 | ACCESS der Directory. Wird kopiert und bei ENTRY & UPDATE wieder zurückübertragen. Eine Veränderung oder Prüfung findet nicht statt. |
| $FE51 | Länge eines Eintrags in dieser Directory ($27). |
| $FE52 | Anzahl der File-Einträge pro Block ($0D). |
| $FE53/54 | Anzahl der aktiven, d.h. nicht freien/gelöschten Einträge innerhalb sämtlicher Blocks dieser Directory. |

Die folgenden Informationen werden einmal (beim „Login" via internes ONLINE) aus dem Key Block der Volume Directory gelesen, in den VCB-Eintrag kopiert und von da aus jeweils neu gesetzt:

| | |
|---|---|
| $FE55/56 | Blocknummer des ersten VBM-Blocks (DiskII: 00 06). |
| $FE58/58 | Gesamtzahl der Blocks auf dem Volume (DiskII: $0118). |
| $FE59 | Unitnummer. |

Für den Eintrag selber existieren folgende Speicherstellen:

| | |
|---|---|
| $FE5A/5B | Blocknummer des Key Blocks der Directory, in der der Eintrag steht. |
| $FE5C/5D | Blocknummer des Blocks der Directory, in dem der Eintrag steht. |
| $FE5E | Eintragsnummer innerhalb dieses Blocks. |
| $FE5F | Storage Type und Namenslänge des Eintrags ($:Stype; $xX:Länge). |
| $FE60..6E | Der Name selber in ASCII, aufgefüllt mit $00 (nicht Leerzeichen). |
| $FE6F | File Type. Wird von MLI nur kopiert, nie verändert oder bearbeitet/berücksichtigt. Ausnahme: Laden des „.SYSTEM"-Programms in REBOOT oder nach dem Boot von ProDOS. Hier wird nach einem File mit File Type $FF (SYS) gesucht. |
| $FE70/71 | Nummer des ersten Blocks im File (kann Master/Index/Datenblock oder der Key Block einer (Sub)directory sein). |
| $FE72/73 | Anzahl der vom File belegten Blocks (Blocks-used) incl. Indexblocks etc. |
| $FE74/75/76 | EOF des Files, nicht unbedingt identisch mit Blocks-used. |
| $FE77..7A | CREATION DATE & TIME – nur von CREATE gesetzt. |
| $FE7B/7C | VERSION min/max (00 00) – wird bei OPEN nicht mehr geprüft. |

| | |
|---|---|
| $FE7D | ACCESS des Files – wird im Gegensatz zu ACCESS der Directory geprüft. |
| $FE7E/7F | AUX-INFO. Wird GET FILE für Volume Directories ausgeführt, landen hier „Länge eines Eintrags"/„Einträge pro Block" und werden entweder nicht berücksichtigt (READ) oder sofort überschrieben (GET FILE INFO). Für Datenfiles steht hier z.B. die Startadresse im Speicher (Maschinenprogramme, BASIC...). |
| $FE80..83 | MODIFICATION DATE & TIME |
| $FE84/85 | Nummer des Key Blocks der Directory, in der dieser Eintrag steht. Diese Information wird bei CREATE angehängt und immer nur wieder kopiert, wenn der Eintrag (ENTRY & UPDATE) neu geschrieben wird. Berücksichtigt oder ausgewertet wird die Information nie, das Programm holt sich die entsprechende Blocknummer aus $FE5A/5B. |

Der Rest der Speicherseite $FE00 sind interne Variablen des MLI:

| | |
|---|---|
| $FE86 | – SET MARK: Differenz zwischen „Position jetzt" und „Position gewünscht" in Blocks * 2. Für SET MARK in Directories ist das die Anzahl der Blocks/2, die übersprungen werden muß.<br>– FLUSH: Temporär für das Addieren des Storage Type zur Namenslänge des Files verwendet. |
| $FE86/87 | – Zähler für die freien Blocks in „Space on Volume" ($E973).<br>– Nummer des nächsten freien Blocks (ALLOCATE NEXT BLOCK: $EAB6). |
| $FE86/87/88 | Ergebnis der Addition von „Position jetzt" und Request Count für READ und WRITE. Wird mit dem EOF verglichen. Für READ wird der Transfer-Zähler entsprechend heruntergesetzt, bei WRITE der EOF entsprechend erhöht. |
| $FE89 | nicht verwendet |
| $FE8A/8B/8C | Momentaner EOF des Files, kopiert aus dem FCB-Eintrag |
| $FE8D/8E/8F | Alte „Position jetzt" im File. Wird von SET MARK aus dem FCB-Eintrag kopiert und vom Error Exit von WRITE verwendet. |
| $FE90 | Temporär im Vergleich „Key Block – VCB-Eintrag" ($E929) verwendet und sonst nirgendwo („Scratch"). |
| $FE91 | VCB-Index ($00/$20/$40/$60/$80..): Basis des momentan benutzten VCB-Eintrags. |
| $FE92 | FCB-Index ($00/$20/$40...): Basis des momentan benutzten FCB-Eintrags. |
| $FE93 | „Freier Eintrag im FCB gefunden"-Flag. $00 – kein freier Eintrag gefunden. Gesetzt von $EFB3 („Scan FCB"), benutzt von OPEN. |

$FE94/95 „Blocks needed" – Anzahl der Blocks, die belegt werden sollen. Gesetzt vor jedem Aufruf von „Space on Volume" ($E973). Bei der Berechnung der Anzahl der belegten Blöcke für GET FILE INFO (Volume Directories) auch als Zwischenergebnis verwendet.

$FE96 Nur innerhalb von SET MARK verwendet: Storage Type des Files, kopiert aus dem FCB-Eintrag.

$FE97 – Von FIND FILE ENTRY als Zähler für die aktiven Einträge innerhalb eines Directory-Blocks verwendet.
– Für OPEN/RENAME/DESTROY: Flag in „Scan FCB" ($EFB3). Gesetzt <> 0, wenn der File als FCB-Eintrag existiert, d.h. OPEN ist.

$FE98/99 Kopie von $FE53/54: Anzahl der aktiven Einträge. Wird in FIND FILE ENTRY beim Absuchen einer Directory nach einem bestimmten Eintrag heruntergezählt (Verifizierung der DIRECTORY STRUCTURE: Es müssen bei der Suche genauso viele Einträge gezählt werden, wie angegeben). Dient innerhalb derselben Routine auch noch als „Indikator" für CREATE.

$FE9A – In FIND FILE ENTRY: (Herunter)zähler für die Einträge innerhalb eines Directory-Blocks, Kopie von $FE52 (Wert: $0D).
– In SET MARK für Directories: Herunter/Heraufzähler für die Anzahl der zu überspringenden Directory-Blocks ($FE86 / 2).
– Für OPEN: Nummer (nicht Index) des ersten freien FCB-Eintrags, der von „Scan FCB" ($EFB3) gefunden wurde.

$FE9B –„Freier Eintrag in der Directory gefunden" – Flag für CREATE, gesetzt von FIND FILE ENTYRY beim Absuchen der Directory ($E6E3).
–In SPACE ON VOLUME ($E973) am Anfang auf $FF („no Space") gesetzt. Wird danach auf den ersten VBM-Extent gesetzt, in dem ein freier Block gefunden wurde ($EA0E) oder bleibt auf $FF => VOLUME FULL.
–In DEALLOCATE BLOCK ($EA34) temporär für die Bitmask ($80/$40/$20/$10/$08..).

$FE9C Herunterzähler in „Space on Volume" ($E973) für die Anzahl der möglichen VBM-Blocks.

$FE9D Start- und momentaner Index in REMOVE BLOCKS für das DEALLOCATE von Blocks, auf die ein Master/Indexblock zeigt. Temporäre Variable, wird innerhalb von REMOVE BLOCKS gesetzt!

$FE9E Gesamtlänge des Pathname (+ Prefix) in Copy & Verify Pathname.
$FE9F PREFIX UNIT. Wird von SET PREFIX auf die entsprechende Unitnummer gesetzt, aber leider nirgendwo benutzt. Das Volume wird bei

einem gesetzten Prefix im VCB gesucht und die Unitnummer aus dem gefundenen VCB-Eintrag geholt. Das ist auch ganz gut so, denn bei $E87B wird diese Speicherstelle immer auf das momentane Unit gesetzt, wenn der Pathname zu Ende ist.

$FEA0/A1 PREFIX DIRECTORY. Wird von SET PREFIX gesetzt und beinhaltet die Blocknummer des Key Blocks der „PREFIXten" Directory. Enthält das Prefix nur einen Namen, haben diese Speicherstellen den Wert 00 02, d.h. sie zeigen auf die Volume Directory. Im Gegensatz zu PREFIX UNIT von FIND FILE ENTRY verwendet.

$FEA2 – In DEALLOCATE BLOCK: Blocknummer/8, d.h. Index innerhalb der VBM-Bytes (Basisadresse).
– In ALLOCATE NEXT BLOCK: Blocknummer/8, d.h. derselbe Zweck. Wird hier allerdings nur gespeichert und nie mehr benutzt.

$FEA3 In ALLOCATE NEXT BLOCK: Blocknummer / $800, d.h. Speicherseite innerhalb eines VBM-Blocks. Gesetzt durch VBM-„Extent" / 2 beim Laden eines VBM-Blocks und während der Suche kontinuierlich erhöht. Wird hinterher zur Berechnung der Nummer eines gefundenen freien Blocks verwendet.

$FEA4 – In DEALLOCATE BLOCK: Speicherseite innerhalb der VBM, d.h. Blocknummer / $800 (s. $FEA3 in ALLOCATE NEXT BLOCK).
– In ALLOCATE NEXT BLOCK: Speicherseite innerhalb des VBM-Blocks. Dient nur als Indikator (0/1), die Blocknummer wird von $FEA3 bestimmt.

$FEA5 „VBM altered". Gesetzt auf $80, wenn der VBM-Block, der sich momentan im Speicher befindet, verändert wurde und zurückgeschrieben werden muß, bevor ein anderer VBM-Block geladen werden kann. Nach dem Zurückschreiben bekommt diese Speicherstelle den Wert $00.

$FEA6 „VBM Unit". Nummer des Units, von dem sich gerade ein VBM-Block im Speicher befindet.

$FEA7/A8 Blocknummer des ersten VBM-Blocks für dieses Unit. Kopie aus dem VCB-Eintrag.

$FEA9 „VBM Extent". Nummer des VBM-Blocks (nicht Blocknummer!), der gerade im Speicher ist. Für den ersten VBM-Block hat diese Speicherstelle den Wert 00, bei einer ProFile nimmt sie Werte zwischen 00 und 02 an (3 VBM-Blocks).

$FEAA/AB/AC – „Position gewünscht" im File für SET MARK.
– SET EOF: neuer EOF, später als „Position gewünscht" für SET MARK verwendet, wenn der File verkleinert wurde.

**$FEAD/AE** Transfer-Herunterzähler für READ und WRITE. Wird durch den Request Count gesetzt und bei READ entsprechend verkleinert, wenn READ den EOF des Files überschreiten würde.

**$FEAF** Anzahl der kompletten Blocks, die noch übertragen werden sollen. Nur von READ (Stufen 2 und 3) verwendet.

**$FEB0** NEWLINE BYTE – nur von READ verwendet. Kopie aus dem FCB-Eintrag.

**$FEB1** NEWLINE MASK – nur von READ verwendet. Kopie aus dem FCB-Eintrag.

**$FEB2** Mystery Flag für READ, Stufe 3. Wird von jedem I/O via SET MARK auf $FF gesetzt, vor Stufe 2 von READ auf $00 (s. „Spezielles").

**$FEB3** $C0-$D3 Command Handler ($E047): „Shift"-Byte für die Routinen, die vor dem eigentlichen Command auszuführen sind. Bekommt den Wert aus der Tabelle auf $FDCD. Je nach gesetzten Bits im Bereich Bit 7-5 werden „Copy & Verify Pathname" ($E08A), „Set File Data" ($E1D0) und SET TIME ($BF06) vorher ausgeführt. Nirgendwo anders verwendet.

**$FEB4** Internes Backup-Flag des MLI (s. „Spezielles").

**$FEB5** In ONLINE ALL: hat ein anderes Unit denselben Volume-Namen und einen offenen File, wird dieses Flag gesetzt. Führt zu ERR#57 DUPLICATE VOLUME.

**$FEB6** In ONLINE ALL: VCB-Index des doppelten Volumes, auf dem ein offener File gefunden wurde.

**$FEB7** $C0-$D3 Command Handler: Index innerhalb der ersten Tabelle ($00-$13). Wird beim Aufruf hier gespeichert und nie wieder benutzt.

**$FEB8** – Copy & Verify Pathname ($E08A): Zähler für die Länge eines Namens im Path. Mit Anfangswert $FF auch gleichzeitig als Indikator „erstes Zeichen im Namen", d.h. besondere Prüfung verwendet.
– ONLINE ($E2E1): Herunterzähler für das Kopieren eines Volume-Namens in den Puffer des „user"-Programms.
– „Scan Directory Block" ($E6E3): Herunterzähler für den Vergleich eines Directory-Eintrags mit dem gesuchten Namen.
– „Scan VCB" ($E7A4): Herunterzähler für Namensvergleich.

**$FEB9** RENAME Files/Subdirectories: Start(index) des unterschiedlichen Namens im neuen Path.

$FEBA Startindex des momentan behandelten Namens im Path. Ausnahme: ONLINE ALL: Start des nächsten Eintrags im Puffer des „user"-Programms.

$FEBB Startindex des (gesamten) Path: Existiert ein System Prefix und ist der angegebene Name nicht vollständig, hat diese Speicherstelle einen Wert im Bereich $C0..$FE, bei einem vollständigen Pathname hat sie den Wert $00.

$FEBC FULL NAME Flag. Hat den Wert $AE (= „/") bei einem vollständigen Namen oder den Wert $00, wenn ein System Prefix vorangestellt wird. Gesetzt von Copy & Verify Pathname.

$FEBD – In Copy & Verify Pathname: Scratch für den momentanen Index im Path.
– In ONLINE ALL: Herunterzähler innerhalb der von der Global Page kopierten DEVICE TABLE.

$FEBE – Zwischenspeicher für Fehler, die bei CLOSE/FLUSH ALL auftreten. Gesetzt durch die Error-Routine auf $F5F7. Wird nach dem Ende des jeweiligen „ALL"-Loops zurückgeladen.

SET EOF und DESTROY benutzen zusammen dieselbe Routine (REMOVE BLOCKS: $FA78). DESTROY setzt dabei die Untergrenze auf 00 02 00, d.h. es bleibt ein Datenblock vom File übrig, der dann danach ebenfalls komplett entfernt wird.
SET EOF setzt beliebige Werte zum Verkleinern eines Files.
REMOVE BLOCKS setzt ggf. den Storage Type des Files herunter und verändert damit die Nummer des „ersten Blocks im File".

$FEBF/C0 „Erster Block im File" für SET EOF/DESTROY. Wird von REMOVE BLOCKS verändert, wenn der Storage Type des Files heruntergesetzt wird.

$FEC1 Storage Type des Files für REMOVE BLOCKS.

$FEC2/C3 Anzahl der von REMOVE BLOCKS freigegebenen Blocks.

$FEC4 Untergrenze für das REMOVE innerhalb des Masterblocks.

$FEC5 Untergrenze für das REMOVE innerhalb eines Indexblocks.

$FEC6/C7 Untergrenze für das REMOVE innerhalb eines Datenblocks. Ist diese Untergrenze < $200, wird innerhalb des Datenblocks gelöscht und der Datenblock danach zurückgeschrieben.

REMOVE BLOCKS benutzt zusätzlich die drei Speicherstellen $FE9D und $FEC8/C9:

$FEC8 Index innerhalb eines Masterblocks für die Kopie von jeweils bis zu acht existierenden Indexblocks.

$FEC9 Herunterzähler innerhalb dieser 8 kopierten Indexblocks.

---

| | |
|---|---|
| $FECA | – bis $FED1: Blocknummern (low) von 8 kopierten Indexblocks für REMOVE BLOCKS.<br>– bis $FED7: aus der Global Page kopierte DEVICE TABLE für ONLINE ALL und internes ONLINE in FIND FILE ENTRY. |
| $FED2 | – bis $FED9: Blocknummern (high) von 8 kopierten Indexblocks für REMOVE BLOCKS. |
| $FEDA/DB | Länge des „user"-Puffers für GET PREFIX ($40), ONLINE ($10, ALL:$100) und READ/WRITE. |
| $FEDC | nicht benutzt |
| $FEDD/DE | Startadresse des Filepuffers, aus der Global Page kopiert. Nur von Set File Data ($E1D0) und SET/GET BUFFER benutzt. |
| $FEDF..$FEEE | IRQ-Handler: Stack Space für den Fall, daß bei einem Interrupt der Stack mehr als halbvoll (Stackpointer < $80) ist. Dieser Bereich ist gegenüber den Versionen 1.0.x verlegt worden. |
| $FEEF..$FEF4 | IRQ Handler: Zwischenspeicher für die Speicherstellen $FA..$FF von der Zeropage. |
| $FEF5/F6 | – Startadresse der Commands $C0-$D3. Wird via indirektem JSR vom Command Handler ($E047) aufgerufen.<br>– Startadresse der RWTB in DO DISK I/O ($DEDA). Wird mit der Adresse aus der DRIVER TABLE besetzt, danach via JSR ($FEB5) Sprung zur RWTB. |
| $FEF7..$FEFF | nicht benutzt |
| --- | --- |
| $FF00..$FF7F | Interface zur RAMDisk-RWTB |
| $FF80..$FFFF | IRQ, NMI und RESET-Handler für den Fall, daß eines dieser Signale auftritt, während die Language Card eingeschaltet ist. |

## 6.2.8. Die RAMDisk-RWTB der Version 1.1.1

Da der IRQ-Vektor bei einem „neuen Monitor" ebenfalls auf AUX geschrieben wird, enthält die RAMDisk einen Block weniger. Der letzte Block ($7F), der auch innerhalb des Speichers die höchste Adresse hat ($FE00..$FFFF in AUX), muß belegt sein, um ein Überschreiben des IRQ-Vektors zu verhindern.
Da der Programmierer von ProDOS allerdings den Test auf „maximale Blocknummer" innerhalb der RWTB auf die ihm eigene Art und Weise gelöst hat, nämlich mit „LDA Blocknummer / BMI ERROR" anstelle eines „CMP Maximum", und dieser Test auch in der

Version 1.1.1 nicht geändert wurde, kann man den IRQ-Vektor mit WRITE BLOCK auf die RAMDisk in die ewigen Jagdgründe schicken.

Da die RAMDisk sich selbst „formatiert", ist die FORMAT-Routine innerhalb der AUX-RWTB entsprechend geändert worden, um den Block $7F auch bei einer frisch „formatierten" RAMDisk zu belegen.

Dieser Zusatz braucht 5 Byte Speicherplatz:
– die drei NOPs auf $0207..$020A wurden gestrichen;
– in FORMAT wird der Block $7F zusätzlich belegt, der restliche Code der AUX-RWTB verschiebt sich um zwei Byte nach hinten;
– im EXIT der AUX-RWTB wurden ebenfalls zwei NOPs (vorher auf $03E8/E9) gestrichen. Damit endet die AUX-RWTB auf der gleichen Adresse wie in den Versionen 1.0.x.

Innerhalb der RWTB-Routinen auf $FF00..$FF80 gibt es nur eine Änderung: der Sprung zum RTS innerhalb der AUX-RWTB via XFER nach einem WRITE von MAIN nach AUX ($FF62..$FF7E) führt nicht mehr nach $02D8, sondern folgerichtig nach $02DA.

```
0200-   AD 18 C0    LDA   $C018          ; Read 80STORE
0203-   48           PHA                  ; Save des Status von 80STORE
0204-   8D 00 C0    STA   $C000          ; Abschalten, das Display bleibt auf MAIN
        ********             ; drei NOPs entfallen
0207-   A2 04       LDX   #$04
0209-   B5 42       LDA   $42,X          ; Kopie der RWTB-Parameter auf $03BD..

020B-   9D BD 03    STA   $03BD,X          in AUX
020E-   CA          DEX
020F-   10 F8       BPL   $0209

0211-   2D BC 03    AND   $03BC          ; INITFLAG: bereits formatiert?
0214-   D0 39       BNE   $024F          ; ja, und COMMAND ist auch nicht FORMAT
        ........
```

– „Formatieren" der VBM

```
0228-   A9 FE       LDA   #$FE      *** ; der letzte Block ist belegt!
022A-   8D D1 03    STA   $03D1     ***

022C-   98          TYA                  ; = $FF
022D-   A0 0E       LDY   #$0E           ; (nicht $0F)
0230-   99 C2 03    STA   $03C2,Y        ; $03C3: FF FF FF FF ... FE
0233-   88          DEY
0234-   D0 FA       BNE   $0230
0236-   8C C2 03    STY   $03C2          ; $03C2 = $00, d.h. Blocks $00..$07 belegt
        .........

03BC-   00                      ; INITFLAG, $FF = „bereits formatiert"

03BD-   00                      ; COMMAND XOR 3
03BE-   00                      ; Unitnummer – nicht benutzt
03BF-   00 00                   ; Ziel/Quelladresse in MAIN
03C1-   00                      ; Blocknummer, später Zieladresse (hi) in AUX
```

„VBM" einer frisch formatierten RAMDisk:
Blocks $00..$07 belegt, Blocks $08..$7E frei, Block $7F ist belegt (IRQ-Adresse)!

```
03C2- 00 FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FE

03D2- F3 52 41 4D                ; Storage Type/Namenslänge und „RAM"
03D6- C3 27 0D 00 00 03 00 7F    ; ACCESS, Eintragslänge.. - kopiert in die VolDIR
```

– EXIT

```
03DE-    AD 8B C0    LDA   $C08B        ; BANKSELECT der LC-Bank 1
03E1-    AD 8B C0    LDA   $C08B        ; + WRITE ENABLE
03E4-    68          PLA
03E5-    10 03       BPL   $03EA
         ****                           ; zwei NOPs gestrichen

03E7-    8D 01 C0    STA   $C001        ; Anschalten von 80STORE
03EA-    4C EF 03    JMP   $03EF        ; Sprung zu $FF44 in MAIN via XFER
         .....
```

## 6.2.9. Interrupts der Version 1.1.1

**A) Die „normale" IRQ-Behandlung**

Der logische Ablauf der IRQ-Behandlung ist derselbe geblieben, die verzwickte Hin- und Herspringerei zwischen MLI und der System Global Page findet auch in der Version 1.1.1 noch statt. Erfreulicherweise sind hier sogar die Adressen konstant geblieben, so daß uns ein erneutes Listing erspart bleibt: der IRQ-Einsprung befindet sich immer noch zusammen mit der RESET-Routine auf $FF9B in der Language Card.

Allerdings ist die Version 1.1.1 auf einen neuen Monitor-ROM hin ausgelegt.
Innerhalb des Relocators wird die IRQ-Startadresse des Monitor-ROMs geprüft. Wenn diese Startadresse unterhalb von $D0xx liegt, also außerhalb des Monitor-ROMs, wird sie in die Language Card und in den AUX-RAM der RAMDisk kopiert. Des weiteren wird im IRQ-„Verteiler" ($DF3A) ein Flag gesetzt: die Speicherstelle $DFD8 bekommt den Wert $01, wenn der „neue Monitor" entdeckt wurde. (IRQ-Startadresse im IIc ist $C803).
Wegen des IRQ-Vektors in AUX enthält die RAMDisk einen Block weniger, der Speicherbereich $FE00..$FFFF innerhalb von AUX ist damit gegen ein Überschreiben aus dem MLI (nicht allerdings gegen WRITE BLOCK) geschützt. Hier muß der Programmierer endlich einmal selbst ausbaden, daß auch innerhalb der RAMDisk-RWTB der Test auf „maximale Blocknummer" nicht mit einem „CMP" stattfindet (s. nächstes Kapitel).

Für einen APPLE II/II+ oder IIe ändert sich im IRQ-Ablauf überhaupt nichts – vorausgesetzt, daß Apple, Inc. nicht den neuen Monitor-ROM in Zukunft in den IIe einbaut.

Was der IRQ-Handler innerhalb des neuen Monitor-ROMs tut, läßt sich ungefähr erahnen (z.Zt. liegt ein neues ROM-Listing nicht vor).
Zum einen dürfte die Speicherstelle $45 hoffentlich vor einem Überschreiben bei einem IRQ „im Monitor" geschützt sein, zum anderen wird hier offensichtlich die IRQ-Returnadresse und das P-Register (Flags) Monitor-intern zwischengespeichert.

Innerhalb der IRQ-Behandlung im MLI ($DF3A) wird das Flag $DFD8 vor dem Zwischenspeichern und dem späteren Zurückladen der IRQ-Returnadresse und des P-Reg getestet. Wenn das Flag gesetzt ist, unterbleiben diese Aktionen. Sonst hat sich hier nichts geändert.

**B) Die „Mystery Routine"**

Diese Routine wird vom Relocator zusammen mit dem MLI in die Language Card kopiert.

```
FD3B-   08          PHP
FD3C-   68          PLA             ; P-Reg in den Akku

FD3D-   48          PHA               und P-Reg wieder auf den Stack
FD3E-   B8          CLV
FD3F-   29 04       AND   #$04      ; Test des I-Flag:
FD41-   F0 03       BEQ   $FD46     ; RTS mit V-Flag gelöscht, wenn IRQ erlaubt
FD43-   2C 64 FD    BIT   $FD64     ; = $60, d.h. setze V-Flag
FD46-   78          SEI
FD47-   AD 83 C0    LDA   $C083     ; Anschalten der LC-Bank 2

FD4A-   AD 83 C0    LDA   $C083       und WRITE ENABLE
FD4D-   38          SEC
FD4E-   A2 05       LDX   #$05
FD50-   20 D6 03    JSR   $03D6     ; kopiert $0042..47 von MAIN nach AUX
FD53-   8D 0F BF    STA   $BF0F       RTS mit #$28 im Akku => SYSERR
FD56-   AD 8B C0    LDA   $C08B     ; Bank 1 der LC
FD59-   AD 8B C0    LDA   $C08B       und WRITE ENABLE

FD5C-   28          PLP             ; Restore des INT-Status
FD5D-   18          CLC
FD5E-   AD 0F BF    LDA   $BF0F     ; hier immer(?) #$28
FD61-   F0 01       BEQ   $FD64
FD63-   38          SEC             ; RTS mit ERR=28: NO DEVICE CONNECTED (?)
FD64-   60          RTS
```

Auf $03D6 wurde zusammen mit den Vektoren für BRK, RESET etc. folgende Routine vom Relocator kopiert:

```
03D6-   8D 08 C0    STA   $C008     ; ALTZP - Select MAIN Zpage/Stack und LC
03D9-   B5 42       LDA   $42,X
03DB-   8D 09 C0    STA   $C009     ; ALTZP - Select AUX Zpage/Stack und LC
03DE-   95 42       STA   $42,X
03E0-   CA          DEX
03E1-   10 F3       BPL   $03D6     ; Loop für $0047..42
03E3-   A9 28       LDA   #$28
03E5-   38          SEC
03E6-   8D 08 C0    STA   $C008     ; ALTZP - Rückschaltung auf MAIN

03E9-   60          RTS
```

und auf $03EA findet sich die Startadresse der „Mystery Routine":

```
03EA-   3B FD
```

Der Rest des Bereichs $03D0..$03FF ist „normal" belegt, d.h. hier gibt es keine Unterschiede zu den Versionen 1.0.x.

Folgende Spekulationen seien hier erlaubt:
Da innerhalb des Relocators keine weiteren Aktionen bezüglich dieser Routinen durchgeführt werden, müßte eine weitere Routine innerhalb des „neuen Monitors" existieren (nicht beim IIc!), die aus dem „LDA #$28" auf $03E3 ein „LDA #00" macht, falls ein entsprechendes DEVICE (welches?) installiert ist.
Des weiteren muß innerhalb dieses Monitors eine Routine existieren, die den Inhalt der Speicherstellen $03EA/EB als Vektor benutzt.

Es ist anzunehmen, daß hier mit den Speicherstellen $0042..47 die RWTB-Parameter kopiert werden. Dazu müßte aber eine neue RAMDisk-RWTB (vielleicht mit einer Extended 80-Col Card mit erweitertem Speicher, z.B. 256k?) geschaffen werden.

Wahrscheinlich wissen wir bei der nächsten Auflage dieser Analyse Genaueres, bis dahin heißt es: abwarten.

# ANHÄNGE

## Anhang A. „Spezielles"

In diesem Kapitel sind alle Merkwürdigkeiten innerhalb des MLI und der DISKII-RWTB zusamengefaßt. Es handelt sich hierbei zuerst um einige echte Fehler seitens des Programmierers von ProDOS, des weiteren um nicht dokumentierte „Features" und schlußendlich um Dinge, die der Autor dieser Analyse nicht kapiert hat.
Speziell die „echten" Programmierfehler gelten nur für die Versionen 1.0.1 bis 1.1.1. Es ist anzunehmen, daß Apple, Inc. diese Fehler entweder schon korrigiert hat – oder sie ohne viel Aufhebens korrigieren wird – und dafür nicht gleich eine neue Versionsnummer „spendiert".

Nachdem die „hot line" von Apple München z.Zt. aus einem jungen Mann besteht, der, obwohl sichtlich bemüht, leider außerstande war, so „exotische Programmierfragen" (Zitat!) wie den Wert einer Monitor-Speicherstelle für den IIe (nötige Aktion: Nachschlagen im entsprechenden Handbuch und Vorlesen) zu beantworten, wurde von vorneherein davon abgesehen, diese Stelle noch einmal zwecks Klärung der hier aufgeworfenen Fragen zu Rate zu ziehen...

### 1) Programmierfehler

$D0C9 – innerhalb des Command Handler für READ und WRITE. Hier ist wahrscheinlich der „dickste Hund" versteckt.
Der dort stattfindende Aufruf zum Test des Quell/Zielgebiets mit der System Bit Map (JSR $EE89) springt **hinter** dem Test auf „<$200 / >$BFFF" in die Prüfroutine ein.
Folge: durch Setzen einer Quell/Zieladresse > $BFFF wird beim Test die System Bit Map überschritten. Die Routine, die aus einer gegebenen Speicherseiten-Nummer den Index innerhalb der SBM berechnet, enthält keinen weiteren Test.
Auf der System Global Page folgt nach der SBM die BUFFER TABLE – bei einer Quell/Zieladresse >$BFFF **wird gegen den Inhalt der BUFFER TABLE „geprüft"**.
Je nachdem, ob Files OPEN sind oder nicht, sind damit einzelne Speicherseiten „belegt" oder „frei". Mit BLOCKWRITE kann man auf diese Weise das MLI selber blockweise auf die Diskette schreiben, bei BLOCKREAD ist die Language Card ebenfalls angeschaltet (und wie bei allen MLI-Funktionen WRITE ENABLEd).
Der „Selbstschutz" gegen ein READ in das Betriebssystem selber hinein ist damit derselbe wie bei DOS 3.3, nämlich nicht vorhanden. Darüberhinaus ist dieser Schutz noch mehr oder weniger „random" und die ganze SBM ist für die Versionen 1.0.1 bis 1.1.1 für die Katz.

Innerhalb des MLI befinden sich zwei Stellen, an denen sich der Programmierer von ProDOS vertippt hat:

$D421 – der Error Exit von „Set File Data“. Wird ein beliebiges Command im Bereich $C0-$D3 gegeben, das sich auf einen OPEN File bezieht, wird vor der Ausführung des eigentlichen Command die Routine „Set File Data“ aufgerufen. Hier wird überprüft, ob der File einen FCB-Eintrag hat, also OPEN ist. Ist das nicht der Fall, verabschiedet sich die Routine mit ERR#43: INVALID REFERENCE NUMBER, wobei der sowieso nicht gültige FCB-Eintrag noch einmal ungültig gemacht werden soll (Byte $00 des Eintrags (Refno/„besetzt“) auf $00:

```
D421-   A9 00       LDA  #$00
D423-   99 52 F0    STA  $F052,Y    ; hier gehört „$F300,Y" hin
D426-   A9 43       LDA  #$43       ; ERR#43: INVALID REFERENCE NUMBER
        ....
```

$F052 ist der FCB-Index, $F300 ist die FCB-Basis. Es ist mit Sicherheit anzunehmen, daß der Programmierer diese beiden Symbole verwechselt hat. Dieser Fehler ist vollständig „redundant“, es werden dabei witzigerweise immer – je nach Reference Number – die Variablen $F052/$F072/$F092/$F0B2... auf Null gesetzt. Alle diese Speicherstellen werden bei jedem MLI-Aufruf neu gesetzt bzw. nie benutzt. Das dürfte auch der Grund sein, warum dieser Bug bei den Tests nicht bemerkt worden ist.

$DE6D – der Error Exit von SET MARK – es wurde versucht, SET MARK in einem File mit Storage Type 0 durchzuführen. Hier sollte ebenfalls der FCB-Eintrag für diesen File ungültig gemacht werden, herausgekommen ist dabei allerdings folgendes:

```
DE6D-   A0 52       LDY  #$52       ; hier gehört „LDY  $F052" hin
DE6F-   99 00 F3    STA  $F300,Y    ; A=00 (Storage Type)
DE72-   A9 43       LDA  #$43       ; ERR#43: INVALID REFERENCE NUMBER
```

$F052 ist der FCB-Index des „momentanen“ Files (s.o.). Hier ist dem Programmierer ein „#“ hineingerutscht und der Assembler hat als „default“ den niederwertigen Adressenteil von $F052, eben den Wert $52 eingesetzt.
Dieser Fehler ist der einzige Programmierfehler, mit dem es wirklich Ärger geben kann: Wenn vier Files gleichzeitig OPEN sind und dieser Fehler passiert, steht Byte $12 des vierten FCB-Eintrags, also die „Position jetzt“-niederwertigstes Byte plötzlich auf Null. So unwahrscheinlich diese Konstellation auch ist – wird auf diesen File danach geschrieben, dann stimmt der Pointer nicht mehr und es werden maximal 255 Bytes mit Daten überschrieben, die eigentlich „weiter hinten“ im File stehen sollten.

Ein dritter Programmierfehler befindet sich im SEEKABS der DISKII-RWTB ($F933). Der Einsprung zu SEEKABS:

```
F933-   8D 72 FB    STA  $FB72     ; „Track gewünscht" * 2
F936-   CD 5A FB    CMP  $FB5A     ; „Track jetzt" * 2
F939-   F0 4C       BEQ  $F987     ; Tracks sind gleich: keine Armbewegung
```

Korrekt wäre hier ein Sprung zum nächsten CLC/RTS (wie in den entsprechenden Routinen von DOS 3.3, UCSD-Bios oder CP/M) – statt dessen wird etwas „zu kurz" gesprungen, nämlich ans Ende von SEEKABS zum ARM MOVER.
Durch den Vergleich ist das Carry jetzt gesetzt. Der ARM MOVER (Phasen-Adressierung des Arm-Motors) schaltet daraufhin eine Phase ON, d.h. es wird versucht, den Arm zu bewegen. Daß das nicht wehtut, liegt nur daran, daß nach dem Exit von SEEKABS (zurück zu MYSEEK) sofort alle Phasen des Arm-Motors OFF geschaltet werden. Diese Phase ist also maximal rund 40 Mikrosekunden lang ON – in dieser Zeit bewegt sich der Motor bestimmt nicht.

## 2) Ein Fehler in der Logik des MLI

Auch dieser Fehler ist äußerst unwahrscheinlich – im Gegensatz zu den oben aufgeführten drei Bugs ist hier allerdings das Ergebnis ein Total-Crash.

Die Speicherstellen $F0B5/B6 sind doppelt belegt: Sowohl der Handler für die $C0-$D3 Commands ($D23C) als auch die Routine „DO DISK I/O" ($D0DA) speichern hier die Startadresse des jeweiligen Command bzw. die Startadresse der RWTB aus der System Global Page. Während in „DO DISK I/O" diese Adresse gespeichert und sofort danach auch dahin gesprungen wird, vergehen beim Handler für die $C0-$D3 Commands u.U. einige Millisekunden – je nachdem, welche Routinen vor dem eigentlichen Command ausgeführt werden. Passiert während dieser Zeit ein Interrupt, erfolgt ein Aufruf des IRQ-Handlers ($D13A).
Hier wird geprüft, ob noch mindestens $80 Bytes auf dem Stack zur Verfügung stehen. Ist das nicht der Fall, werden 16 Bytes vom Stack heruntergeholt und zwischengespeichert. Der Zwischenspeicherbereich ist $F0AF bis $F0BE, d.h. die Speicherstellen $F0B5/B6 werden dabei überschrieben.
Nach dem Abarbeiten der Service-Routine wird der Stack zurückgeladen, die Programmausführung geht im Handler weiter, bis schließlich das eigentliche Command aufgerufen wird – via „JMP ($F0B5)". Wo der Prozessor dabei landet, läßt sich dann nicht einmal raten.

## 3) Undokumentiertes & Unklares in Reihenfolge der Speicheradressen

$0804 – auf Sektor 00/00 innerhalb des Boot: Hier steht ein „JMP $A132", der wahrscheinlich in den APPLE III-ROM führt – oder auch nicht. Frage: wie kommt der Prozessor auf $0804?

$0932 – innerhalb des Blocks 00 00 (Boot). Hier wird die Returnadresse $A000 auf den Stack gebracht, danach werden X-A mit 00 01 (Blocknummer?) geladen. Mit einem Sprung nach $F479 verschwindet die Ausführung im Dunklen. Frage: wenn innerhalb des APPLE III-ROM eine entsprechene RWTB steht, die danach Block 00 01 liest $- von woher wird $0932 angesprungen?

$2062 – Test der Speicherstelle $FBC0 (F8-ROM) und entsprechendes Setzen von MACHID im Relocator. Frage: welche APPLE-Maschinen außer dem IIc kommen da noch auf uns zu?

$25B8 – innerhalb der Slot-Suchroutine des Relocators. Wenn MACHID immer noch $00 ist, wird hier erneut auf „APPLE II“ getestet. Ein Eintrag in MACHID findet nicht statt, falls eine Uhrenkarte gefunden wurde. Dasselbe gilt für die erweiterte 80-Zeichen-Karte. MACHID wurde allerdings vorher schon beim Test auf 48/64k <> 00 gesetzt. Frage: ist das eine zusätzliche „Sicherung“, um Leute zu ärgern, die nur teilweise hinter den „Kompatiblen-Test“ gekommen sind?

$27B0 – innerhalb des MOVER im Relocator. X-Y werden mit der Startadresse des nächsten Parametersatzes geladen, danach folgt RTS mit Carry gesetzt. $27B0 wird von nirgendwoher angesprungen. Von welcher Version ist dieses Programmstück stehengeblieben und welche Funktion hatte es?

$BF94 – „Level“ – wird bei OPEN in den FCB-Eintrag (Byte $1C) kopiert und von CLOSE (nur bei „ALL“) getestet. Innerhalb des MLI wird $BF94 nicht gesetzt oder verändert. Da in APPLESOFT ebenfalls kein Befehl enthalten ist, der eine CLOSE-Hierarchie von Files setzt, bleibt die Natur dieses Bytes vorläufig im Dunklen. Vielleicht ergibt die demnächst vom selben Autor folgende „Analyse des BASIC.SYSTEM“ neue Erkenntnisse.

$BF95 – „BUBIT“ – im MLI existiert ein internes Backup-Bit ($F074). Es wird bei jedem Command im Bereich $C0-$D3 erst einmal gesetzt ($D255: $F074 <= $20). Verändert wird $F074 nur in SET FILE INFO ($E9D4).
Wenn der ACCESS des Files das Backup-Bit gesetzt hat und gleichzeitig das BUBIT ($BF95) nicht gesetzt ist, wird $F074 auf $20 gesetzt, ansonsten bekommt diese Speicherstelle den Wert $00.
Benutzt wird $F074 schließlich in ENTRY & UPDATE ($D6BB): es findet ein ORA des ACCESS mit $F074 statt, d.h. wenn das interne Backup-Bit gesetzt war, ist das Backup-Bit im ACCESS des File-Eintrags damit auf jeden Fall gesetzt – und bleibt es auch – ein Löschen dieses Bits in ACCESS ist im MLI außer durch SET FILE INFO nicht möglich. Man darf gespannt sein, welches zukünftige Utility dieses Bit verwendet.

$D888 – „Verifizierung“ eines Subdirectory-Headers
Das Byte $14 eines Subdirectory Headers (erster Eintrag im Key Block) muß fünf gesetzte Bits enthalten, ansonsten folgt ERR#4E: INCOMPATIBLE FILE FORMAT. Abgesehen davon, daß diese „Prüfung“ recht merkwürdig ausgefallen ist – welche Erweiterungen sind da noch geplant bzw. welchen Zweck hat das Ganze?

$E107 – OPEN
Das Command OPEN sucht zuerst einmal den File-Eintrag auf dem Volume via GET FILE, bevor der FCB nach dem (gefundenen) File abgesucht wird. Die Suchroutine im FCB ($E1A9) bricht mit Carry gesetzt ab, wenn der File im FCB gefunden wurde, d.h. bereits OPEN ist und OPEN kehrt wieder mit ERR#50: FILE (ALREADY) OPEN – aber nur, wenn der File im FCB-Eintrag nicht auf READ ONLY gesetzt war.

Existiert der File dagegen im FCB, wird der FCB-Eintrag neu gesetzt. Ist wiederum nur READ erlaubt (für Subdirectories ist das immer der Fall), passiert gar nichts, die Ausführung von OPEN geht weiter mit dem Setzen und Belegen des Filepuffers etc.
Ist dagegen WRITE ebenfalls erlaubt, wird jetzt erst geprüft, ob der File schon als READ ONLY im FCB existiert hat (Flag „File im FCB gefunden" ($F057), gesetzt von $E1A9). Ist das der Fall, folgt Abbruch mit ERR#50: FILE OPEN. Der Merkwürdigkeiten nicht genug: da der gesamte FCB-Eintrag vor dem Neusetzen gelöscht wurde, ist der neu erstellte Eintrag damit ungültig – Byte $00 (Refno/„besetzt") hat den Wert $00.
Abgesehen davon, daß es mit der vorliegenden Version des MLI nicht möglich ist, einen File via OPEN zu eröffnen, danach den ACCESS zu verändern und den bereits offenen File ein weiteres Mal zu eröffnen – welchen Zweck sollte dieser Programmteil ursprünglich erfüllen?

$E2A7 – READ, Stufen 2 und 3 – das „Mystery Flag" $F072
Vor der „Stufe 2", dem Setzen der SET MARK-Pointer auf den Zielbereich, wird $F072 auf $00 gesetzt. Jeder tatsächliche I/O von SET MARK aus (hier: Lesen vom Volume – nicht Löschung innerhalb des Puffers, weil nicht existierender Block) setzt $F072 wieder auf $FF, d.h. $F072 <> $00 bedeutet: „vom Volume gelesen".
Innerhalb des Loops von „Stufe 3" wird $F072 wieder auf $00 zurückgesetzt, wenn ein „sparse"-Block, d.h. ein nicht existierender Datenblock im File steht. (Dieser Block muß ein „sparse"-Block sein, da der Transfer-Zähler vorher auf den EOF des Files begrenzt worden ist – es kann sich also nicht um einen Block „hinter" dem EOF handeln).
Der Loop in „Stufe 3" springt solange zum SET MARK-Aufruf (d.h. „Stufe 2") zurück, bis wieder ein Datenblock von SET MARK tatsächlich vom Volume gelesen wurde. Erst dann wird direkt aus dem Indexblock bestimmt und via DO DISK I/O gelesen.
Logisch begründbar ist die Notwendigkeit des Aufrufs von SET MARK für einen „sparse"-Block. Hier muß der entsprechende Bereich im Zielgebiet gelöscht werden (und genau das tut SET MARK).
Warum allerdings mindestens ein „echter" SET MARK-Aufruf dem direkten Lesen eines existierenden Blocks vorausgehen muß, ist (mir) unklar.

$E3B1 – READ für Directories
Der Transfer Count wurde, genauso wie für Datenfiles nötigenfalls auf den EOF begrenzt. Nach Abschluß des Loops, in dem solange SET MARK aufgerufen wird, bis der gesamte Transfer abgewickelt ist, wird (via dem gemeinsamen Ende von READ/WRITE) die Routine SET MARK ein weiteres Mal aufgerufen. SET MARK verzweigt intern zu „SET MARK in Directory Files" ($DFAE), wo in diesem Fall via Forward Reference gearbeitet wird.
Wurde die Directory von READ vollständig gelesen (bis zum EOF), dann steht die „Position gewünscht" für SET MARK jetzt auf dem nächsten Block, d.h. ein Byte hinter dem EOF. In diesem Fall versucht SET MARK, die Forward Reference zum nächsten Directory-Block zu bestimmen und bricht mit ERR#4C: EOF ENCOUNTERED ab (die Forward Reference im letzten Directory-Block hat den Wert 00 00).
Nach dem RTS zu READ (Directories) wird geprüft, ob ERR#4C vorliegt, wenn nicht, folgt RTS zu $D078.
Der ERR#4C wird maskiert, schließlich wurde die READ-Aktion ja fehlerfrei beendet. Die Nummer des „Datenblocks momentan im Speicher" wird im FCB-Eintrag auf 00 00 gesetzt – es befindet sich kein Datenblock der Directory mehr im Filepuffer. Der Datenteil des Filepuffers wird gelöscht – wozu? Jeder weitere Aufruf von READ führt zu einem EOF

ENCOUNTERED und Abbruch, bevor überhaupt zwischen 'Datenfile' und 'Directory' unterschieden wird.
Noch erstaunlicher ist allerdings, wo die Nummer des letzten Blocks der Directory landet: in den beiden ersten Bytes des Datenteils des Filepuffers nämlich. Was diese Blocknummer dort zu suchen hat, ist unklar – allerdings wird sie auch nie mehr von dort gelesen.

$E627 – Erhöhung des Storage Type: Bit 3 im STATUS des Files wird gesetzt. Dieses Bit wird nirgendwo getestet oder ausgewertet, eine eventuelle spätere Erweiterung des MLI, in der dieses Bit genutzt werden könnte, ist auch nicht ersichtlich.

$EC71 – REMOVE BLOCKS
Wenn REMOVE BLOCKS für einen File mit Storage Type <> 0/1/2/3 aufgerufen wurde, folgt SYSTEM DEATH mit einer recht merkwürdigen Fehlernummer, nämlich dem dem Storage Type des Files * 16.

## 4) Erwähnenswertes außerhalb des MLI

Der FILER hat gleich zwei merkwürdige Eigenschaften:

a) Beim Formatieren einer Diskette besteht der Address Field Trailer nicht aus DE AA EB, sondern aus DE AA DE EB. Hier hat sich also ein zusätzliches Byte eingeschlichen. Dieses Byte wird nirgendwo innerhalb des FILER oder anderer ProDOS-Programme wieder gelesen/verifiziert, es nimmt aber zusätzlichen Platz auf einem Track ein. Die Folge ist u.U. die Meldung „DRIVE TOO FAST" beim Formatieren auf einem Laufwerk, das DOS 3.3 oder andere Betriebssysteme klaglos verarbeitet. Programmierer, die mit „zeitkritischen" Kopierschutzsystemen arbeiten, werden ebenfalls wenig Freude daran haben – der Track ist mit diesen zusätzlichen 16 Bytes wirklich randvoll.

b) Das mysteriöse „$00-Command" für die RWTB sollte – wenn man dem Programmierer des FILER glauben darf – noch eine weitere Funktion haben.
Zur Feststellung von TOTAL BLOCKS (Gesamtkapazität) eines Volumes hat der FILER gleich mehrere Möglichkeiten:
1) Test auf DISKII und Festlegung auf $118 Blocks;
2) Lesen der Speicherstellen $CxFD/FE auf der Controller-Karte;
3) Sind auch diese Speicherstellen unbesetzt (00 00) und ist das Laufwerk keine DISKII, gibt der FILER ein „$00-Command" an die RWTB weiter und erwartet als Return TOTAL BLOCKS in A-Y !

# Anhang B – Patches für das EXER.SYSTEM und den FILER

Nachdem sich gezeigt hat, daß Apple, Inc. keine Mühe scheut, die Speicheradressen innerhalb des MLI inkonstant zu halten, ist eine Beschreibung von Patches für das MLI selber ziemlich sinnlos.
Anders verhält es sich mit den Zusatzprogrammen unter ProDOS.

## 1) Patch für das EXER.SYSTEM

Apple, Inc. hat mit dem EXER.SYSTEM ein Programm zum Experimentieren mit dem MLI herausgebracht. Mit diesem Programm kann man einzelne MLI-Befehle ausgeben und das Resultat unmittelbar beobachten, ohne selber gleich zum Assembler greifen zu müssen. Vor die Beobachtung, was ein Befehl MLI-intern bewirkt, haben die Apple-Mannen allerdings einen recht dicken Riegel geschoben – die Funktion MODIFY BUFFER des EXER.SYSTEM, mit der man einzelne Speicherbereiche auf den Schirm holen und verändern kann, zeigt das APPLESOFT-ROM anstelle der Bank 1 der Language Card.
Das Herausziehen der Language Card funktionert zwar im Prinzip, weil das EXER.SYSTEM auch unter 48k lauffähig ist – da das MLI aber bei dieser Speichergröße die VBM der Zieldiskette demoliert, sollte man das lieber lassen (s. „Unterschiede zu 48k-Systemen").

Der folgende Patch, der diesen unerfreulichen Zustand beseitigt, wird hier nur „zum Eintippen" aufgeführt. Eine ausführliche Beschreibung des „quid, quod et quomodo" erscheint in einem Artikel der Zeitschrift PEEKER (Hüthig-Verlag Heidelberg).

Allerdings sollte an dieser Stelle darauf hingewiesen werden, daß Apple, Inc. auch mittlerweile verschiedene Versionen des BASIC.SYSTEM herausgebracht hat, ohne die Versionsnummern zu ändern. Falls Ihr EXER.SYSTEM z.B. mit den Befehlen $82 und $65 etwas anfangen kann, ist größte Vorsicht geboten – die Patches gelten für (die angeblich einzige) Version des EXER, die auf diese Befehle überhaupt nicht reagiert.

Die erforderlichen Patches „Step-by-Step":

– Kopie der EXERCISER-Disk mit dem FILER

UNLOCK EXER.SYSTEM
BLOAD EXER.SYSTEM, A$2000, TSYS
CALL -151

```
33A9: 20 EF 3C EA EA
33B2: A5 09
3CEF: 8D 82 C0 20 C1 FB AD 8B C0 B1 DD 8D 82 C0 85 09 60
32DE: 48 AD 8B C0 AD 8B C0 68
32E6<32E9.3323M
3321: 8D 82 C0
32ED: 3A
3356: 20 E0 3C EA
```

```
3CE0: AC 8B C0 AC 8B C0 A4 DF 91 DD 8C 82 C0 60
3AC0: 3C
```

BSAVE EXER.SYSTEM, A$2000, L$1D00, TSYS

Wenn alles funktioniert, können Sie danach MLI-Internes nicht nur beobachten, sondern auch verändern. Die Funktion (U)ndo arbeitet ebenfalls korrekt. Die Meldung „PROTECTED MEMORY PAGE" wird zwar nach wie vor ausgegeben, eine Veränderung wird aber nicht mehr gesperrt.

## 2. Patch für den FILER

Auch dieser Patch erscheint in einem PEEKER-Artikel und ist hier nur als „Step-by-Step" angegeben. Er ermöglicht das Formatieren kompatibler Laufwerke mit mehr als 35 Tracks – vollständige DISKII-Kompatibilität vorausgesetzt: mit einem 160-Spur Drive und/oder anderem Pre/Denibbling wird man an dem FILER keine rechte Freude haben.

– Kopie des FILER mit dem FILER selber

UNLOCK FILER
BLOAD FILER, A$2000, TSYS
CALL -151

```
79F4: 28   ; Anzahl der Tracks, hier 40

4244: 40   ; Anzahl der Blocks (low)
4246: 01   ; Anzahl der Blocks (hi), ergibt zusammen $140 anstelle $118 Blocks
                                     für 40 Tracks (1 Track = 8 Blocks)
```

BSAVE FILER, A$2000, L$6400, TSYS
RENAME FILER, FILER40

Danach folgt eine Korrektur der DISKII-RWTB:

UNLOCK PRODOS
BLOAD PRODOS, A$2000, TSYS
CALL -151

Für die Version 1.0.1 kann man den Test auf „maximale Blockanzahl" überspringen mit:

```
5209: D0 05 EA ; wird nach dem Relocate zu „BNE .. / NOP"
```

Für die Version 1.0.2 wird der Aufruf der Testroutine durch ein „CLC" ersetzt:

```
5201: 18 EA EA ; „CLC / NOP / NOP" anstelle von „JSR $FEBE"
```

BSAVE PRODOS, A$2000, L$3D00, TSYS

Ein RENAME ist hier ohne entsprechende Änderung des Boot (Block 00 00, Speicherstellen im Urlader: $0902...) nicht möglich, weil der Urlader sonst den File nicht mehr findet.

# Anhang C. Directory-Struktur und File-Einträge

Zur Verdeutlichung der Directory-Struktur sind hier die Blocks der im Kapitel „VCB vs FCB" postulierten Diskette DEMO gelistet, in denen sich diese Diskette von einer frisch formatierten Disk unterscheidet. Hier wird angenommen, daß die beschriebenen Experimente bereits beendet sind, d.h. der Textfile DATEN2 wurde durch WRITE um einen Datenblock erweitert und umfaßt jetzt drei Blocks (1 Index- und 2 Datenblocks).

„CAT /DEMO" zeigt:

/DEMO
NAME TYPE BLOCKS MODIFIED
DATEN1 BAS 1 <NO DATE>
SUBDIR DIR 1 <NO DATE>

BLOCKS FREE: 268 BLOCKS USED: 12

„CAT /DEMO/SUBDIR" zeigt:

SUBDIR
NAME TYPE BLOCKS MODIFIED
DATEN2 TXT 3 <NO DATE>

BLOCKS FREE: 268 BLOCKS USED: 12

Der erste Block der Volume Directory (Block 00 02) hat jetzt insgesamt drei Einträge: den Header mit dem Volume-Namen DEMO sowie die Einträge für DATEN1 und SUBDIR. Die restlichen Blocks der Volume Directory sind unverändert.

```
000- 00 00 03 00 F4 44 45 4D  4F 00 00 00 00 00 00 00    ....tDEMO.......
010- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00    ................
020- 00 00 C3 27 0D 02 00 06  00 18 01 16 44 41 54 45    ..C'........DATE
030- 4E 31 00 00 00 00 00 00  00 00 00 FC 07 00 01 00    N1.........ü....
040- 25 00 00 00 00 00 00 00  00 E3 01 08 00 00 00 00    %........c......
050- 02 00 D6 53 55 42 44 49  52 00 00 00 00 00 00 00    ..VSUBDIR.......
060- 00 00 0F 08 00 01 00 00  02 00 00 00 00 00 00 00    ................
070- E3 00 00 00 00 00 00 02  00 00 00 00 00 00 00 00    c...............
080- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00    ................
090- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00    ................
0A0- ................
```

Im VBM-Block (Block 00 06) sind jetzt die Blocks 00 07 bis 00 0B ebenfalls belegt.

```
000- 00 0F FF FF FF FF FF FF  FF FF FF FF FF FF FF FF    ................
010- FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF    ................
020- FF FF FF 00 00 00 00 00  00 00 00 00 00 00 00 00    ................
030- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00    ................
040- ................
```

Block 00 07 enthält das BASIC-Programm DATEN1. Dieses Programm besteht nur aus einer Zeile (Nummer 10) mit einem REM-Statement (Token B2), der Rest des Blocks ist leer.

```
000- 24 08 0A 00 B2 20 48 49  45 52 20 49 53 54 20 44    $...2 HIER IST D
010- 41 53 20 50 52 4F 47 52  41 4D 4D 20 44 41 54 45    AS PROGRAMM DATE
020- 4E 31 00 00 00 00 00 00  00 00 00 00 00 00 00 00    N1..............
030- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00    ................
040- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00    ................
050- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00    ................
060- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00    ................
070- ................
```

Block 00 08 enthält den Key Block von SUBDIR, der hier auch der einzige Block der Subdirectory ist. In diesem Block steht der Header als erster Eintrag, danach folgt der File-Eintrag für DATEN2.

```
000- 00 00 00 00 E6 53 55 42  44 49 52 00 00 00 00 00    ....fSUBDIR.....
010- 00 00 00 00 75 48 55 53  54 4F 4E 21 00 00 00 00    ....uHUSTON!....
020- 00 00 C3 27 0D 01 00 02  00 03 27 26 44 41 54 45    ..C'......' & DATE
030- 4E 32 00 00 00 00 00 00  00 00 00 04 0A 00 03 00    N2..............
040- 06 02 00 00 00 00 00 00  00 E3 00 00 00 00 00 00    .........c......
050- 08 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00    ................
060- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00    ................
070- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00    ................
080- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00    ................
090- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00    ................
0A0- ................
```

Block 00 09 ist der erste **Daten**block des Files DATEN2. Da im Beispiel mit dem zweiten WRITE die Größe des Storage Type 1 überschritten wurde, ist dieser Block nicht mehr identisch mit dem „ersten Block im File", was auch im File-Eintrag ersichtlich ist (hier im Key Block von SUBDIR die Bytes $3C/$3D. Sie zeigen auf Block 00 0A).
Im Beispiel wurde dieser Block vollständig mit „XXXX" gefüllt:

```
000- 58 58 58 58 58 58 0D 58  58 58 58 58 58 58 58 58    XXXXXX.XXXXXXXXX
010- 58 58 58 58 58 58 58 58  58 58 58 58 58 58 58 58    XXXXXXXXXXXXXXXX
020- 58 58 58 58 58 58 58 58  58 58 58 58 58 58 58 58    XXXXXXXXXXXXXXXX
030- ................
1D0- ................
1E0- 58 58 58 58 58 58 58 58  58 58 58 58 58 58 58 58    XXXXXXXXXXXXXXXX
1F0- 58 58 58 58 58 58 58 58  58 58 58 58 58 58 58 58    XXXXXXXXXXXXXXXX
```

Block 00 0A enthält den Indexblock von DATEN2. Der File bestand ursprünglich nur aus einem Datenblock (Block 00 09); nachdem durch WRITE das Fassungsvermögen dieses Blocks überschritten wurde, wurde zuerst ein Indexblock und danach ein weiterer Datenblock belegt.
Dieser Indexblock enthält die Blocknummern der beiden Datenblocks, der höherwertige Teil der Blocknummer ist im Beispiel in beiden Fällen 00.

```
000- 09 0B 00 00 00 00 00 00  00 00 00 00 00 00 00 00    ................
010- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00    ................
020- ................
100- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00    ................
110- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00    ................
120- ................
```

Block 00 0B enthält den Rest des Files DATEN2. Da der File im Beispiel nicht mit PRINT : PRINT CHR$(4) „CLOSE" geschlossen wurde, steht hinter dem letzten „X" auch kein <CR>.

```
000- 58 58 58 58 58 58 00 00  00 00 00 00 00 00 00 00    XXXXXX..........
010- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00    ................
020- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00    ................
030- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00    ................
040- ................
```

Auf dieser Beispiel-Diskette finden sich damit außer einem File mit Storage Type 3 alle möglichen Directory-Einträge und Block-Arten.
Der folgende Abriß der Struktur der einzelnen Einträge bezieht sich deshalb einfacherweise auf die Einträge der Diskette /DEMO.

## 1) Volume Directory

Diese Directory steht immer auf den Blocks 00 02 bis 00 05. Im Gegensatz zu Subdirectories ist das Anhängen eines weiteren Blocks nicht möglich.

Innerhalb jedes Volume Directory Blocks sind die ersten vier Byte für die Backward bzw. Forward Reference reserviert.
Die Backward Reference (Bytes $00/$01) enthält die Blocknummer des vorhergehenden Blocks der Directory – 00 00 bedeutet: dies hier ist der erste Block.
Die Forward Reference (Bytes $02/$03) enthält die Blocknummer des nächsten Blocks der Directory – 00 00 bedeutet: dies hier ist der letzte Block.
Diese beiden „Links" wären prinzipiell nicht nötig, weil die Volume Directory immer aufeinanderfolgende Blocknummern belegt. Sie existieren deshalb, weil die Suchroutinen in diesem Punkt keinen Unterschied zwischen Volume und Subdirectories machen (s. dort).

Der erste Block innerhalb der Volume Directory ist der Key Block.
Der erste „Eintrag" innerhalb dieses Key Blocks ist der Header der Volume Directory. Dieser Eintrag beginnt mit Byte $04 des Key Blocks, die Byte-Nummern sind relativ zum Beginn des Eintrags angegeben.

```
00- F4
01- 44 45 4D 4F 00 00 00 00 00 00 00 00 00 00 00   („DEMO")
10- 00 00 00 00 00 00 00 00
18- 00 00 00 00
1C- 00 00
1E- C3
1F- 27
20- 0D
21- 02 00
23- 06 00
25- 18 01
```

Byte $00 – Storage Type/Namenslänge. Der Storage Type steht im höherwertigen Nibble und ist für diesen Eintrag immer $0F ($F0). Die Namenslänge ist hier 4.

Bytes $01-$0F – der Name der Volume Directory, d.h. des Volumes selber in ASCII, das MSB ist gelöscht. Der Name kann maximal 15 Zeichen haben. Ist bei einem RENAME der

neue Name kürzer als der alte, werden die restlichen Zeichen nicht auf $00 gelöscht – Vergleiche arbeiten immer mit der Namenslänge als Zähler.

Bytes $10-$17 – unbenutzt

Bytes $18-$1B – FORMAT DATE & TIME – im selben Format gehalten wie alle DATE & TIME-Einträge. Wird bei FORMAT durch den FILER gesetzt und kann via GET FILE INFO gelesen werden.

Bytes $1C/$1D – ProDOS Version und Min-Version. Nur das Byte $1D wird bei einem OPEN getestet – hat es für die Versionen 1.0.1 und 1.0.2 nicht den Wert $00, folgt Abbruch mit ERR#4E: INCOMPATIBLE FILE FORMAT.

Byte $1E – ACCESS der Directory. Dieser Wert wird nie ausgewertet oder verändert. Die Entscheidung, ob WRITE oder SET EOF nach einem OPEN zulässig ist, erfolgt über den Storage Type.

Byte $1F – Länge eines File-Eintrags innerhalb dieser Directory.

Byte $20 – Anzahl der File-Einträge pro Directory-Block.

Bytes $21/$22 – Anzahl der „aktiven", d.h. existierenden Einträge innerhalb der gesamten Volume Directory.

Bytes $23/$24 – Blocknummer des ersten VBM-Blocks.

Byte $26/$27 – TOTAL BLOCKS, die Gesamtzahl der Blocks auf diesem Volume.

## 2) Subdirectories

Im Gegensatz zu einer Volume Directory hat eine Subdirectory eine beliebige Anzahl von Blocks. Wenn eine Subdirectory mit CREATE (BASIC.SYSTEM und MLI-Befehl) erstellt wird, bekommt sie einen einzigen Block. Werden innerhalb dieser Subdirectory mehr als $0D File-Einträge erstellt, wird innerhalb von CREATE (nur im MLI) ein weiterer Block an die Subdirectory angehängt.
Im Gegensatz zur Volume Directory sind hier die Forward und Backward References unbedingt nötig: aufgrund des eben beschriebenen Prozesses ist es unmöglich, daß die Blocks einer Subdirectory hintereinander stehen (fortlaufende Blocknummern).

Die Bytes $00-$03 jedes Subdirectory-Blocks enthalten die Backward und Forward References (s. Volume Directory).
Der erste Block einer Subdirectory ist der Key Block dieser Subdirectory.
Der erste „Eintrag" im Key Block ist der Header der Subdirectory. Dieser Eintrag beginnt auf Byte $04 des Blocks. Auf der Beispiel-Diskette DEMO steht ein Subdirectory Key Block auf Block 00 08.

```
00- E6
01- 53 55 42 44 49 52 00 00 00 00 00 00 00 00 00   („SUBDIR")
10- 75 48 55 53 54 4F 4E 21                        („uHUSTON!")
18- 00 00 00 00
1C- 00 00
1E- C3
1F- 27
20- 0D
21- 01 00
23- 02 00
25- 03
26- 27
```

Byte $00 – Storage Type(höheres Nibble)/Namenslänge. Der Storage Type ist immer $0E ($E0) für einen Subdirectory Header, die Namenslänge ist hier 6.

Bytes $01-$0F – der Name der Subdirectory (maximal 15 Zeichen) in ASCII.

Bytes $10-$17 – laut „Technical Manual" reserviert. „HUSTON!" wird nie geprüft oder benutzt, das Byte $10 des Eintrags (Wert: $75) wird „verifiziert" (s. „Spezielles").

Bytes $18-$1B – CREATION DATE & TIME. Wird bei Erstellung der Subdirectory gesetzt und ist auch via GET FILE INFO nicht zurückzuholen – alle FILE INFOs beziehen sich auf den File-Eintrag für die Subdirectory, nicht auf den Header selber (Ausnahme: RENAME und DESTROY, hier wird auch im Header modifiziert).

Bytes $1C/$1D – MLI-Version und Min-Version. Auch hier wird nur Byte $1D geprüft. Es muß für die ProDOS-Versionen 1.0.1 und 1.0.2 den Wert $00 haben.

Byte $1E – ACCESS der Subdirectory – wird nie geprüft oder verändert.

Byte $1F – Länge eines Eintrags innerhalb der Subdirectory.

Byte $20 – Anzahl der File-Einträge pro Subdirectory-Block.

Bytes $21/$22 – Gesamtzahl der aktiven (d.h. nicht gelöschten) File-Einträge, die in dieser Subdirectory verzeichnet sind.

Bytes $23/$24 – der Parent Pointer. Hier steht die Nummer des Blocks, in dem die Subdirectory selber als File-Eintrag verzeichnet ist („Parent Block").

Byte $25 – Eintragsnummer der Subdirectory im Parent Block.

Byte $26 – Länge eines Eintrags in der Parent Directory, also der Directory, in der die Subdirectory selber als File-Eintrag verzeichnet ist.

## 3) File-Einträge

Für Volume Directories kann logischerweise kein File-Eintrag in einer anderen Directory stehen.

Eine Subdirectory ist dagegen genauso wie ein „normales" File als Eintrag verzeichnet, allerdings mit Storage Type $0D (nicht zu verwechseln mit dem Eintrag im Subdirectory Key – hier ist das Erkennungsbyte $0E!).

Ein File-Eintrag („normal" oder Subdirectory) hat immer dasselbe Format – egal, ob er in einer Volume Directory oder in einer Subdirectory steht.

Wir nehmen als Beispiel den File-Eintrag von DATEN2 in der Subdirectory SUBDIR (Block 00 08 auf der Beispiel-Diskette).

```
00- 27
01- 26 44 41 54 45 4E 32 00 00 00 00 00 00 00 00    („DATEN2")
10- 04
11- 0A 00
13- 03 00
15- 06 02 00
18- 00 00 00 00
1C- 00 00
1E- E3
1F- 00 00
21- 00 00 00 00
25- 08 00
```

Byte $00 – Storage Type (höheres Nibble)/Namenslänge. Der Storage Type ist für „normale" Files im Bereich 01..03 ($10..$30), für eine Subdirectory $0D ($D0).
Bei der Löschung des Eintrags wird dieses Byte auf 00 gesetzt.

Bytes $01-$0F – der Name des Files in ASCII.

Byte $10 – der File Type (hier TXT) wird vom MLI nicht ausgewertet (Ausnahme: $FF für SYS-Files beim Boot/REBOOT). Eine Subdirectory hat den File Type $0F.

Bytes $11/$12 – Nummer des „ersten Blocks im File". Zeigt hier auf den Indexblock des Files DATEN2, bei Subdirectories steht hier die Nummer des Key Blocks.

Bytes $13/$14 – BLOCKS USED, die Gesamtzahl der Blocks, die vom File belegt werden, bzw. die Anzahl der Blocks einer Subdirectory.

Bytes $15-$17 – EOF des Files. Bei Subdirectories wird hier die Blockanzahl * $200 eingetragen.

Bytes $18-$1B – CREATION DATE & TIME.

Bytes $1C/$1D – ProDOS-Version und Min-Version. Nur Byte $1D wird überprüft.

Byte $1E – ACCESS des Files. Bei dem File DATEN2 sind RENAME, DESTROY sowie READ und WRITE erlaubt, das mysteriöse Backup-Bit ist ebenfalls gesetzt.

Bytes $1F/$20 – AUX-INFO: für Subdirectories und sequentielle Textfiles unbesetzt, bei einem BAS-File steht hier z.B. die Startadresse (vgl. Eintrag für DATEN1). Innerhalb des MLI wird diese Information nicht benutzt oder ausgewertet, sondern nur weitergegeben.

Bytes $21-$24 – MODIFICATION DATE & TIME

Bytes $25/$26 – hier steht die Blocknummer des Key Blocks der Directory, in der der File-Eintrag steht. Da DATEN2 in der Subdirectory SUBDIR steht, ist hier Block 00 08 angegeben. Bei DATEN1 z.B. steht hier Block 00 02, weil sich dieser File-Eintrag in der Volume Directory befindet.

## 4) „Block-Arten"

### 4.1) Der VBM-Block

In diesem Block befindet sich das Verzeichnis der freien/belegten Blocks eines Volumes (vgl. VTOC von DOS 3.3). Überschreitet das Fassungsvermögen eines Volumes die Grenze von $1000 Blocks, werden beim FORMAT mehrere VBM-Blocks angelegt (die ProFile hat z.B. drei VBM-Blocks).
Im Gegensatz zu allen anderen Strukturen unter ProDOS existiert hier keine Reference der Blocks untereinander – die Nummer des ersten VBM-Blocks steht im Header der Volume Directory verzeichnet, weitere VBM-Blocks folgen (wenn nötig) lückenlos in aufsteigender Reihenfolge. Bei der ProFile besteht die VBM aus den Blocks 00 06, 00 07 und 00 08.

**4.2) Directories** – s. Punkte A-C.

### 4.3) Der Masterblock

Ein File mit Storage Type 3 enthält einen (einzigen) Masterblock, in dem maximal 128 Indexblocks verzeichnet sein können. Der Grund für diese Begrenzung liegt in der Verwendung von 24 Bit für EOF und Position innerhalb eines Files. Die größte mit 24 Bit darstellbare Position ist 256 * 256 * 256. Dadurch ergibt sich eine maximale Filegröße von 128 * 256 * 512, also 128 Indexblocks, die ihrerseits jeweils auf 256 Datenblocks mit 512 Byte Inhalt zeigen.
Ein Masterblock ist „optisch" nicht von einem Indexblock zu unterscheiden, die Blocknummern sind in derselben Art angeordnet.

### 4.4) Der Indexblock

Ein Indexblock enthält maximal 256 Blocknummern von Datenblocks eines Files. Ein File mit Storage Type 2 enthält einen (einzigen) Indexblock, ein File mit Storage Type 3 enthält maximal 128 Indexblocks.
Die Blocknummern der Datenblocks sind innerhalb eines Indexblocks seitenweise angeordnet, d.h. die niederwertige Hälfte der Nummer eines Blocks steht in den ersten $100 Byte, die höherwertige Hälfte in den zweiten $100 Byte mit demselben Index innerhalb der Speicherseite. Der erste Datenblock, auf den ein Indexblock zeigt, belegt also die Bytes $00 und $100,

der zweite Datenblock die Bytes $01 und $101 innerhalb eines Indexblocks. Der Indexblock der Files DATEN2 auf der Beispiel-Diskette zeigt auf die Datenblocks 00 09 und 00 0B.

### 4.5) Der Datenblock

Ein Datenblock umfaßt 512 Bytes und stellt die kleinste Menge an Information dar, die unter ProDOS standardmäßig gelesen/geschrieben werden kann. Der Inhalt ist beliebig, es existieren keinerlei Referenzen oder Querverbindungen zu anderen Datenblocks.
Wenn ein File via DESTROY gelöscht wird, bleiben alle Datenblocks des Files unverändert auf dem Volume erhalten, sie werden lediglich in der VBM wieder freigegeben. Index/Masterblocks werden dagegen tatsächlich gelöscht (s. DESTROY/SET EOF).
Unbeschriebene Teile eines Datenblocks enthalten Nullen (also nicht z.B. $ 1A wie CP/M-Records).

## 5) „Verschiedenes"

### 5.1) ACCESS

Dieses Byte ist in Directories zwar definiert, kann aber dort nicht verändert werden.
Innerhalb eines File-Eintrags (und damit auch für Subdirectories) haben die einzelnen Bits folgende Bedeutung:

Bit 7 – DESTROY – Wenn gesetzt, ist DESTROY erlaubt. Bei Subdirectories wird allerdings zuerst geprüft, ob keine File-Einträge innerhalb dieser Subdirectory vorhanden sind. Ist die Subdirectory nicht leer, ist DESTROY nicht möglich.

Bit 6 – RENAME – wenn gesetzt, ist RENAME erlaubt. Gilt für Subdirectories und „normale" Files. Für Volume Directories wird anstelle des ACCESS aus dem Header immer $C3 eingesetzt (FIND FILE ENTRY), d.h. RENAME ist hier vom ACCESS her immer möglich.

Bit 5 – BACKUP BIT – wird vom BASIC.SYSTEM benutzt (s. „Spezielles").

Bit 4/3/2 – „reserviert" und unbenutzt.

Bit 1 – WRITE – wenn gesetzt, ist WRITE erlaubt. Gilt nur für „normale" Files, bei (Sub)directories wird dieses Bit immer maskiert und intern auf Null gesetzt.

Bit 0 – READ – wenn gesetzt, ist READ erlaubt. Ein Versuch, einen File zu lesen, der dieses Bit in ACCESS nicht gesetzt hat, führt zu ERR#4E: FILE ACCESS ERROR und Abbruch.

### 5.2) Storage Types

Eine Zusammenfassung aller Storage Types, die momentan definiert sind:

$0F – Volume Directory, Erkennungsbyte im Header;
$0E – Subdirectory, Erkennungsbyte im Header;
$0D – Subdirectory, Storage Type im File-Eintrag;
$03 – File mit einem Masterblock, Storage Type im File-Eintrag;
$02 – File mit einem Indexblock, Storage Type im File-Eintrag;
$01 – File mit einem einzigen Datenblock, Storage Type im File-Eintrag;
$00 – Gelöschter File, Storage Type im File-Eintrag.

# Anhang D. Blockbelegung einer mit dem FILER formatierten Diskette

**Block 00 00:**

Die erste Hälfte dieses Blocks enthält den ProDOS-Urlader (im Kapitel „Boot/Reboot" beschrieben). Im Gegensatz zu DOS 3.3 steht dieser Urlader nicht irgendwo im MLI, sondern ist nur ein Teil des FILER.

```
000- 01 38 B0 03 4C 32 A1 86  43 C9 03 08 8A 29 70 4A   .80.L2!.CI...)pJ
010- 4A 4A 4A 09 C0 85 49 A0  FF 84 48 28 C8 B1 48 D0   JJJ.X.I .H(H1HP
020- 3A B0 0E A9 03 8D 00 08  E6 3D A5 49 48 A9 5B 48   :0.)....f=%IH)ÄH
030- 60 85 40 85 48 A0 63 B1  48 99 94 09 C8 C0 EB D0   ".X.H c1H...HP
040- F6 A2 06 BC 1D 09 BD 24  09 99 F2 09 BD 2B 09 9D   v„.<..=$..r.=+..
050- 7F 0A CA 10 EE A9 09 85  49 A9 86 A0 00 C9 F9 B0   .J.n)..I). .Iy0
060- 2F 85 48 84 60 84 4A 84  4C 84 4E 84 47 C8 84 42   /.H.".J.L.N.GH.B
070- C8 84 46 A9 0C 85 61 85  4B 20 12 09 B0 68 E6 61   H.F)..a.K ..0hfa
080- E6 61 E6 46 A5 46 C9 06  90 EF AD 00 0C 0D 01 0C   fafF%FI..o-.....
090- D0 6D A9 04 D0 02 A5 4A  18 6D 23 0C A8 90 0D E6   Pm).P.%J.m#.(..f
0A0- 4B A5 4B 4A B0 06 C9 0A  F0 55 A0 04 84 4A AD 02   K%KJ0.I.pU ..J-.
0B0- 09 29 0F A8 B1 4A D9 02  09 D0 DB 88 10 F6 29 F0   .).(1JY..PÄ..v)p
0C0- C9 20 D0 3B A0 10 B1 4A  C9 FF D0 33 C8 B1 4A 85   I P; .1JIP3H1J.
0D0- 46 C8 B1 4A 85 47 A9 00  85 4A A0 1E 84 4B 84 61   FH1J.G)..J ..K.a
0E0- C8 84 4D 20 12 09 B0 17  E6 61 E6 61 A4 4E E6 4E   H.M ..0.fafa$NfN
0F0- B1 4A 85 46 B1 4C 85 47  11 4A D0 E7 4C 00 20 4C   1J.F1L.G.JPgL. L
100- 3F 09 26 50 52 4F 44 4F  53 20 20 20 20 20 20 20   ?.&PRODOS
110- 20 20 A5 60 85 44 A5 61  85 45 6C 48 00 08 1E 24     %".D%a.E1H...$
120- 3F 45 47 76 F4 D7 D1 B6  4B B4 AC A6 2B 18 60 4C   ?EGvtWQ6K4.&+."L
130- BC 09 A9 9F 48 A9 FF 48  A9 01 A2 00 4C 79 F4 20   <.).H)H).„.Lyt
140- 58 FC A0 1C B9 50 09 99  AE 05 88 10 F7 4C 4D 09   Xö .9P.....wLM.
150- AA AA AA A0 D5 CE C1 C2  CC C5 A0 D4 CF A0 CC CF   *** UNABLE TO LO
160- C1 C4 A0 D0 D2 CF C4 CF  D3 A0 AA AA AA A5 53 29   AD PRODOS ***%S)
170- 03 2A 05 2B AA BD 80 C0  A9 2C A2 11 CA D0 FD E9   .*.+*=.X),„.JPüi
180- 01 D0 F7 A6 2B 60 A5 46  29 07 C9 04 29 03 08 0A   .Pw&+"%F).I.)...
190- 28 2A 85 3D A5 47 4A A5  46 6A 4A 4A 85 41 0A 85   (*.=%GJ%FjJJ.A..
1A0- 51 A5 45 85 27 A6 2B BD  89 C0 20 BC 09 E6 27 E6   Q%E.'&+=.X <.f'f
1B0- 3D E6 3D B0 03 20 BC 09  BC 88 C0 60 A5 40 0A 85   =f=0. <.<.X"%X..
1C0- 53 A9 00 85 54 A5 53 85  50 38 E5 51 F0 14 B0 04   S)..T%S.P8eQp.0.
1D0- E6 53 90 02 C6 53 38 20  6D 09 A5 50 18 20 6F 09   fS..FS8 m.%P. o.
1E0- D0 E3 A0 7F 84 52 08 28  38 C6 52 F0 CE 18 08 88   Pc .R.(8FRpN...
1F0- F0 F5 BD 8C C0 10 FB 00  00 00 00 00 00 00 00 00   pu=.X.ä.........
```

**Block 00 01**

Dieser Block enthält den Urlader des SOS-Betriebssystems. Ein Analyseversuch ohne die entsprechenden APPLE III-Handbücher erschien mir sinnlos – wie man sieht, „verschwindet" das Programm erst einmal im ROM (JMP $A06E).

```
000- 4C 6E A0 53 4F 53 20 42  4F 4F 54 20 20 31 2E 31   Ln SOS BOOT  1.1
010- 20 0A 53 4F 53 2E 4B 45  52 4E 45 4C 20 20 20 20    .SOS.KERNEL
020- 20 53 4F 53 20 4B 52 4E  4C 49 2F 4F 20 45 52 52    SOS KRNLI/O ERR
030- 4F 52 08 00 46 49 4C 45  20 27 53 4F 53 2E 4B 45   OR..FILE 'SOS.KE
040- 52 4E 45 4C 27 20 4E 4F  54 20 46 4F 55 4E 44 25   RNEL' NOT FOUND%
050- 00 49 4E 56 41 4C 49 44  20 4B 45 52 4E 45 4C 20   .INVALID KERNEL
060- 46 49 4C 45 3A 00 00 0C  00 1E 0E 1E 04 A4 78 D8   FILE:.........$xX
070- A9 77 8D DF FF A2 FB 9A  2C 10 C0 A9 40 8D CA FF   )w._„ä.,.X)X.J
```

```
080- A9 07 8D EF FF A2 00 CE  EF FF 8E 00 20 AD 00 20   )..o„.No.. -.
090- D0 F5 A9 01 85 E0 A9 00  85 E1 A9 00 85 85 A9 A2   Pu)..")..a)...)„
0A0- 85 86 20 BE A1 E6 E0 A9  00 85 E6 E6 86 E6 86 E6   .. >!f").ff.f.f
0B0- E6 20 BE A1 A0 02 B1 85  85 E0 C8 B1 85 85 E1 D0   f >! .1.."Hl..aP
0C0- EA A5 E0 D0 E6 AD 6C A0  85 E2 AD 6D A0 85 E3 18   j%"Pf-l .b-m .c.
0D0- A5 E3 69 02 85 E5 38 A5  E2 ED 23 A4 85 E4 A5 E5   %ci..e8%bm#$.d%e
0E0- E9 00 85 E5 A0 00 B1 E2  29 0F CD 11 A0 D0 21 A8   i..e .1b).M. P!(
0F0- B1 E2 D9 11 A0 D0 19 88  D0 F6 A0 00 B1 E2 29 F0   1bY. P..Pv .1b)p
100- C9 20 F0 3E C9 F0 F0 08  AE 64 A0 A0 13 4C D4 A1   I p>Ipp..d  .LT!
110- 18 A5 E2 6D 23 A4 85 E2  A5 E3 69 00 85 E3 A5 E4   .%bm#$.b%ci..c%d
120- C5 E2 A5 E5 E5 E3 B0 BC  18 A5 E4 6D 23 A4 85 E2   Eb%eec0<.%dm#$.b
130- A5 E5 69 00 85 E3 C6 E6  D0 95 AE 4F A0 A0 1B 4C   %ei..cFfP..O  .L
140- D4 A1 A0 11 B1 E2 85 E0  C8 B1 E2 85 E1 AD 66 A0   T! .1b."H1b.a-f
150- 85 85 AD 67 A0 85 86 20  BE A1 AD 68 A0 85 85 AD   ..-g .. >!-h ..-
160- 69 A0 85 86 AD 00 0C 85  E0 AD 00 0D 85 E1 20 BE   i ..-...."-...a >
170- A1 A2 07 BD 00 1E DD 21  A0 F0 08 AE 64 A0 A0 13   !„.=..Ü! p..d  .
180- 4C D4 A1 CA 10 ED A9 00  85 E7 E6 E7 E6 86 E6 86   LT!J.m)..gfgf.f.
190- A6 E7 BD 00 0C 85 E0 BD  00 0D 85 E1 A5 E0 D0 04   &g=...."=...a%"P.
1A0- A5 E1 F0 06 20 BE A1 4C  8A A1 18 AD 6A A0 6D 08   %ap. >!L.! -j m.
1B0- 1E 85 E8 AD 6B A0 6D 09  1E 85 E9 6C E8 00 A9 01   ..h-k m...ilh.).
1C0- 85 87 A5 E0 A6 E1 20 79  F4 B0 01 60 AE 32 A0 A0   ..%"&a yt0 ".2
1D0- 09 4C D4 A1 84 E7 38 A9  28 E5 E7 4A 18 65 E7 A8   .LT!.g8)(egJ.eg(
1E0- BD 29 A0 99 A7 05 CA 88  C6 E7 D0 F4 AD 40 C0 4C   =) .'.J.FgPt-X
1F0- EF A1 00 00 00 00 00 00  00 00 00 00 00 00 00 00   o!..............
```

**Block 00 02**

Der erste Block der Volume Directory hat bei einer frisch formatierten Diskette bereits einen Eintrag, den „Header". Hier ist der Volume-Name, die Anzahl der auf dem Volume verfügbaren Blocks (TOTAL BLOCKS) und CREATION DATE & TIME (hier nicht benutzt) enthalten. Die Backward Reference (Bytes $00/$01) zeigt auf 00 00, d.h. dies hier ist der erste Block. Die Forward Reference (Bytes $02/$03) zeigt auf Block 00 03, d.h. der nächste Block der Volume Directory ist Block 0003.

```
000- 00 00 03 00 F7 42 4C 41  4E 4B 30 30 00 00 00 00   ....wBLANK00....
010- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
020- 00 00 C3 27 0D 00 00 06  00 18 01 00 00 00 00 00   ..C'............
030- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
040- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
050- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
060- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
070- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
080- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
090- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
0A0- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
0B0- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
0C0- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
0D0- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
0E0- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
0F0- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
100- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
110- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
120- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
130- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
140- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
150- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
160- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
170- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
180- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
190- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
```

```
1A0- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
1B0- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
1C0- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
1D0- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
1E0- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
1F0- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
```

**Block 00 03**

Der zweite Block der Volume Directory. Bei einer frisch formatierten Diskette enthält dieser Block nur Backward und Forward Reference und ist ansonsten leer.

```
000- 02 00 04 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
010- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
020- ................
```

**Block 00 04**

Der dritte Block der Volume Directory enthält ebenfalls nur Backward und Forward Reference.

```
000- 03 00 05 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
010- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
020- ................
```

**Block 00 05**

Der vierte und letzte Block der Volume Directory enthält nur die Backward Reference zum vorhergehenden Block. Die Forward Reference ist 00 00, d.h. die Volume Directory ist hier zu Ende.

```
000- 04 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
010- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
020- ................
```

**Block 00 06**

Der einzige VBM-Block für Standard-Disketten (143kB). Die Blocks 00 00 bis 00 06 sind bereits belegt.
Dieser eine VBM-Block könnte $1000 Blocks enthalten, für die DISKII existieren $118 Blocks. Deshalb sind alle möglichen Blocks ab Blocknummer $118 (Zählung ab Blocknummer 00 00) als belegt eingetragen und können vom MLI auch nicht freigegeben werden (Test gegen TOTAL BLOCKS in der Routine DEALLOCATE BLOCK).

```
000- 01 FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
010- FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
020- FF FF FF 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
030- 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
040- ................
```

**Blocks 00 07 bis 01 17**

Diese Blocks sind auf einer frisch formatierten Diskette leer.

# Anhang E. MLI-Reference – Kurzbeschreibung der Parametersätze

Der Aufruf eines MLI-Commands ist von Apple, Inc. folgendermaßen definiert:

```
     JSR  $BF00      ; nur diese Einsprungadresse ist erlaubt
     DFB  COMMAND    ; das Byte COMMAND und die 16-Bit Adresse PARMBLOCK
     DW   PARMBLOCK    werden MLI-intern übersprungen (Returnadresse + 4 / RTI)
     BEQ  NOERROR    ; RTS mit Carry clear und Z-Flag/Akku = 00 wenn kein Fehler
ERR: ...

PARMBLOCK:DFB PARMCOUNT
          DFB/DW PARM1
          DFB/DW ....
```

PARMBLOCK kann irgendwo im Speicher stehen. Das erste Byte steht für die Anzahl der Parameter, die nicht mit der Anzahl der benutzten Bytes identisch sein muß, denn ein Parameter kann zwei oder drei Bytes umfassen.

Die folgende Zusammenstellung entstammt mehr oder weniger dem „Technical Manual" und ist mit zusätzlichen Erläuterungen versehen worden.
Das erste Byte ist in einem Aufruf als COMMAND einzusetzen, die folgenden Bytes stehen im jeweiligen PARMBLOCK für diesen Aufruf.

MLI-intern wird zuerst das COMMAND überprüft. Bei einem nicht definierten COMMAND folgt ERR#01: BAD SYSTEM CALL NUMBER. Danach wird die Anzahl der Parameter (Parmcount) geprüft, bei Nichtübereinstimmung folgt ERR#04: BAD SYSTEM CALL PARAMETER COUNT. Diese beiden Fehlermöglichkeiten sind in der folgenden Aufstellung nicht mehr berücksichtigt.

Des weiteren gibt es ein paar „allgemeine" Fehler, die sich wie folgt einteilen lassen und ebenfalls nicht mehr eigens erwähnt werden:

1) Jeder Suchprozeß nach einem File-Eintrag beinhaltet außer RWTB-Fehlern wie ERR#28: NO DEVICE CONNECTED und ERR#27: I/O ERROR die Fehlermöglichkeiten ERR#44/45/46: DIRECTORY/VOLUME/FILE NOT FOUND.
2) Jedes Update der Directory-Struktur und jedes (Re)WRITE eines Blocks beinhaltet die RWTB-Fehlermöglichkeit ERR#2B: DISK WRITE PROTECTED.
3) Jede Prüfung eines Pathname, bei dem irgendetwas nicht den „Vorschriften" entspricht (ein Name länger als 15 Zeichen, Namensbeginn nicht mit Buchstabe etc.) endet mit ERR#40: INVALID PATHNAME.
4) Alle MLI-COMMANDS, die einen Suchprozeß nach dem File-Eintrag einleiten oder ein Update der Directory-Struktur durchführen (wobei ebenfalls gesucht wird), können potentiell mit Fehlernummern wie ERR#51: DIRECTORY STRUCTURE DAMAGED, ERR#52: NOT A PRODOS VOLUME oder ERR#4B: UNSUPPORTED STORAGE TYPE zurückkehren. Solange Sie allerdings nicht mit BLOCKWRITE an der falschen Stelle herumgespielt oder die Diskette in der Hosentasche transportiert haben, sind diese Fehler reichlich unwahrscheinlich.

5) Alle MLI-COMMANDS, die sich auf eine (File) Reference Number beziehen, prüfen diese Reference Number (Ausnahme: „ALL" (Refno = 00) für CLOSE UND FLUSH). Ist die Refno außerhalb des Bereichs 1-8 oder der entsprechende FCB-Eintrag unbesetzt (kein File mit dieser Refno OPEN), folgt Abbruch mit ERR#43: INVALID REFERENCE NUMBER.

**$40: ALLOC INTERRRUPT**
Parmcount = 2
Reference Number – 1 Byte Resultat
Service-Start – 2 Byte-Adresse
Die Startadresse wird auf den nächsten freien Platz der INTERRUPT TABLE in der System Global Page eingetragen, die Reference Number im Bereich 1-4 wird zurückgeliefert. Diese Refno wird nur für DEALLOC INTERRUPT gebraucht.
Mögliche Fehler: ERR#25: INTERRUPT TABLE FULL (wenn bereits alle vier möglichen Interrupt Service-Startadressen belegt wurden) sowie ERR#53: INVALID SYSTEM CALL PARAMETER (Startadresse des einzutragenden Service kleiner als $100).

**$41: DEALLOC INTERRUPT**
Parmcount = 1
Reference Number – 1 Byte
In der INTERRUPT TABLE auf der System Global Page wird die Startadresse der Interrupt Service-Routine mit der entsprechenden Refno gelöscht. Eine Überprüfung, ob diese Startadresse jemals besetzt war, findet nicht statt.
Einzig möglicher Fehler: ERR#43: INVALID REFERENCE NUMBER – tritt auf, wenn die Reference Number nicht im Bereich 1-4 liegt.

**$65: REBOOT**
Parmcount = 4
.....
REBOOT wird aus der Bank 2 der Language Card kopiert und ausgeführt. Dieses COMMAND ist „inoffiziell" und nachträglich eingebaut worden. Die vier Parameter für dieses COMMAND sind „Dummies", d.h. ihr Wert ist beliebig.
Mögliche Fehler gibt es hier für den MLI-Aufruf selber keine. Bei diesem COMMAND ist Vorsicht geboten: in einem 48k-System erfolgt ein schlichtes RTS innerhalb des MLI und damit die Rückkehr zum „user"-Programm ohne jede Warnung.

**$80: READ BLOCK**
Parmcount = 3
Unitnummer – 1 Byte (DSSS 0000)
Zielpuffer – 2 Byte-Adresse
Blocknummer – 2 Byte
Vom Unit wird der entsprechende Block ab „Zielpuffer" in den Speicher gelesen.
Vor dem Aufruf der entsprechenden RWTB wird der Zielbereich gegen die SBM geprüft: ist die Zieladresse auf einer Seitengrenze, werden zwei, ansonsten drei Speicherseiten geprüft. Bei Zieladressen, die größer als $BFFF sind, ergibt dieser Test Unsinn (s. „Spezielles").
Mögliche Fehler sind ERR#56: BAD BUFFER ADDRESS (Zielgebiet innerhalb der SBM belegt), ERR#28: NO DEVICE CONNECTED (Unitnummer eines nicht existierenden

belegt), ERR#28: NO DEVICE CONNECTED (Unitnummer eines nicht existierenden Devices) sowie Fehler innerhalb der RWTB selber.

**$81: WRITE BLOCK**
Parmcount = 3
Unitnummer – 1 Byte (DSSS 0000)
Quellpuffer – 2 Byte-Adresse
Blocknummer – 2 Byte
Auf das Unit wird der entsprechende Block mit dem Speicherinhalt ab „Quellpuffer" geschrieben.
Vor dem Aufruf der entsprechenden RWTB wird der Quellbereich gegen die SBM getestet: ist die Quelladresse auf einer Seitengrenze, werden zwei, ansonsten drei Speicherseiten geprüft. Bei Zieladressen, die größer als $BFFF sind, ergibt sich derselbe Unsinn wie bei READ BLOCK (s. „Spezielles"). Ein Grund für diesen Test ist für WRITE BLOCK nicht ersichtlich.
Die möglichen Fehler sind dieselben wie bei READ BLOCK, hier kommt noch ERR#2B: DISK WRITE PROTECTED innerhalb der RWTB dazu.

**$82: SET TIME**
Parmcount = 0
Dieses COMMAND ist inoffiziell und bewirkt innerhalb des MLI einen Aufruf von $BF06 – hier sollte dann ein Sprung zur Uhrenroutine (Standard: $F142) stehen. Innerhalb der Uhrenroutine sollten die Speicherstellen $BF90..$BF93 mit DATE & TIME gesetzt werden.
Mögliche Fehler: keine, bei nicht installierter Uhrenkarte steht auf $BF06 ein RTS.

**$C0: CREATE**
Parmcount = 7
Pathname – 2 Byte-Adresse
Access – 1 Byte
File Type – 1 Byte
Aux Type – 2 Byte
Storage Type – 1 Byte
Creation Date – 2 Byte
Creation Time – 2 Byte
Achtung: Dieses COMMAND ist nicht identisch mit dem CREATE des BASIC.SYSTEM!
Mit diesem COMMAND werden Files erstellt. Die Definition von Access, File Type und Aux Type erfolgt nach den gültigen Regeln für diese Parameter (s. Anhang C).
Der Parameter „Pathname" enthält wiederum eine Adresse, auf der der Pathname selber in ASCII stehen muß, also z.B.:

```
PARMBLOCK:  DFB PARMCOUNT
            DW PATHNAME
            ...
PATHNAME:   DFB PATHLENGTH              ; Länge des folgenden Pathname
            ASC '/DEMO/SUBDIR/DATEN2'
```

Der letzte Name im angegebenen Path ist der Name des zukünftigen Files.
CREATE enthält zwei Besonderheiten:

1) Die Unterscheidung 'Create Datenfile' und 'Create Subdirectory' hängt einzig und allein vom angegebenen Storage Type ab, für $0D wird ein Subdirectory Key Block erstellt, für die Storage Types 0(!)/1/2/3 wird ein Datenfile erstellt, der immer den Storage Type 1 hat.
2) Wenn DATE & TIME angegeben sind (<> 00), werden die Werte aus dem Parameterblock benutzt, ansonsten wird die System Time eingesetzt.
Mögliche Fehler: ERR#40: INVALID PATHNAME, ERR#44/45: DIRECTORY/VOLUME NOT FOUND (hier z.B., wenn entweder SUBDIR oder /DEMO nicht gefunden werden konnten), ERR#47: DUPLICATE FILENAME (ein File dieses Namens existiert bereits), ERR#48: VOLUME FULL, ERR#49: VOLUME DIRECTORY FULL, ERR#4B: UNSUPPORTED STORAGE TYPE (für CREATE mit Storage Types <> 0/1/2/3/$D) etc.

**$C1: DESTROY**
Parmcount = 1
Pathname – 2 Byte-Adresse
Der File mit dem Namen „Pathname" wird aus der Directory entfernt, die zum File gehörigen Blocks werden wieder freigegeben. Zur Definition des Pathname im PARMBLOCK siehe CREATE.
DESTROY benutzt dieselbe Routine wie SET EOF zur Freigabe von Blocks. Aus diesem Grund werden Master- und Indexblock(s) des Files tatsächlich gelöscht, der Inhalt der Datenblocks bleibt erhalten.
DESTROY ist in zwei Fällen nicht möglich:
1) Das Access-Byte im File-Eintrag hat das entsprechende Bit (Bit 7) nicht gesetzt.
2) Der File ist eine Subdirectory und enthält seinerseits weitere File-Einträge.
In beiden Fällen folgt ERR#4E: FILE ACCESS ERROR.
Eine Fehlermöglichkeit außer den „allgemeinen" Fehlern ist ERR#50: FILE IS OPEN, d.h. im Gegensatz zu DOS 3.3 können OPEN Files nicht entfernt werden.

**$C2: RENAME**
Parmcount = 2
Pathname – 2 Byte-Adresse
Neuer Pathname – 2 Byte-Adresse
RENAME ist sowohl für Volumes als auch für Subdirectories und Datenfiles möglich. Zur Definition der Pathnamen in PARMBLOCK s. CREATE.
Der Unterschied zwischen neuem und altem Pathname muß im letzten Namen beider Paths auftreten, ansonsten folgt ERR#40: INVALID PATHNAME. Alle anderen Namen der Paths müssen gleich sein. (Es ist allerdings folgende obskure Konstruktion möglich: PREFIX /DEMO, RENAME mit „alter Path": SUBDIR/DATEN2 (unvollständig), „neuer Path": /DEMO/SUBDIR/DATENXXX).
Dieses COMMAND unterscheidet zwischen Volumes und Subdirectories/Datenfiles:
Für Volumes ist RENAME nur möglich, wenn auf dem Volume kein File OPEN ist, ansonsten folgt ERR#50: FILE IS OPEN. Wenn die Volume Directory selber als File geöffnet wurde (für READ ist das möglich), erfolgt Abbruch mit demselben Fehler.
Für Subdirectories/Datenfiles wird nicht nur auf OPEN geprüft, sondern auch, ob der Access des Files das entsprechende Bit (Bit 6) gesetzt hat. Ist das nicht der Fall, folgt ERR#4E: FILE ACCESS ERROR.

Neben den „allgemeinen" Fehlermöglichkeiten wird noch vor dem RENAME geprüft, ob nicht bereits ein File mit dem neuen Namen existiert, ggf. folgt ERR#47: DUPLICATE FILENAME.

**$C3: SET FILE INFO**
Parmcount = 7
Pathname – 2 Byte-Adresse
Access – 1 Byte
File Type – 1 Byte
Aux Type – 2 Byte
Null-Field – 3 Byte
MOD DATE – 2 Byte
MOD TIME – 2 Byte
Dieses COMMAND verändert den File-Eintrag, d.h. die Eigenschaften eines Files. Daraus folgt auch, daß SET FILE INFO für Volume Directories nicht angewendet werden kann, hier folgt ERR#40: INVALID PATHNAME (gesetzt durch die Suchroutine FIND FILE ENTRY und in SET FILE INFO nicht maskiert).
Mit diesem COMMAND kann z.B. der Access eines Files neu gesetzt werden – das Sperren eines Files gegen ein SET FILE INFO ist nicht möglich.
Für Subdirectories kann man damit einen Haufen Unsinn anfangen – nach Veränderung des File Type von $0F auf z.B. $04 sind File-Einträge in einer Subdirectory für CATALOG erst einmal verschwunden. Das MLI selber läßt den File Type außer acht, d.h. die Bearbeitung von Files wird dadurch nicht beeinflußt.
Das Null-Field kann hier beliebige Werte annehmen. Es existiert lediglich, weil für GET FILE INFO dasselbe Format des PARMBLOCK benutzt wird.
Wenn MOD DATE & TIME spezifiziert sind, werden diese Angaben benutzt, ansonsten wird die System Time eingesetzt.
Außer den „allgemeinen" gibt es hier keine weiteren speziellen Fehler.

**$C4: GET FILE INFO**
Parmcount = 7
Pathname – 2 Byte-Adresse
Access – 1 Byte Resultat
File Type – 1 Byte Resultat
Aux Type – 2 Byte Resultat / TOTAL BLOCKS
Storage Type – 1 Byte Resultat
BLOCKS USED – 2 Byte Resultat / TOTAL BLOCKS USED
MOD DATE – 2 Byte Resultat
MOD TIME – 2 Byte Resultat
CREATE DATE – 2 Byte Resultat
CREATE TIME – 2 Byte Resultat
Dieses COMMAND stellt die Inversion zu SET FILE INFO dar. Es wird zwischen Volume Directories und Subdirectories/Datenfiles unterschieden:
Für Volume Directories wird anstelle des Aux Type die Gesamtkapazität des Volumes (TOTAL BLOCKS) und anstelle der Anzahl der von der Volume Directory belegten Blocks (immer 00 04) die Gesamtzahl der auf dem Volume belegten Blocks (TOTAL BLOCKS USED) zurückgeliefert.

Für Subdirectories/Datenfiles enthält das Aux Type Field tatsächlich den Aux Type und BLOCKS USED gibt die Anzahl der vom File belegten Blocks an.
Das Null-Field von SET FILE INFO wird hier mit Storage Type und BLOCKS USED belegt – diese beiden Parameter sind mit SET FILE INFO nicht veränderbar.
Spezielle Fehlermöglichkeiten außer den „allgemeinen" gibt es keine.

**$C5: ON LINE**
Parmcount = 2
Unitnummer – 1 Byte (DSSS 0000) oder 00 („ALL")
Zielpuffer – 2 Byte-Adresse
ON LINE schreibt den Volume-Namen des adressierten Units ab der Adresse „Zielpuffer" in den Speicher. Wenn die Unitnummer mit 00 angegeben ist, werden sämtliche Units aus der DEVICE TABLE auf der System Global Page der Reihe nach bearbeitet. Gleichzeitig findet MLI-intern ein Update des VCB statt.
Der Zielbereich wird vor der Ausführung des COMMAND gegen die SBM geprüft: bei einer spezifizierten Unitnummer (<> 00) werden $10, ansonsten $100 Byte als Platzbedarf angenommen. Ist der entsprechende Speicherbereich in der SBM nicht frei, folgt ERR#56: BAD BUFFER ADDRESS.
Sonstige Fehlermöglichkeiten außer den „allgemeinen" sind hier ERR#52: NOT A PRODOS VOLUME und ERR#55: VOLUME CONTROL BLOCK TABLE FULL (bei mehr als 8 DEVICES) sowie ERR#57: DUPLICATE VOLUME (wenn zwei Volumes denselben Namen haben und mindestens auf einem der beiden Volumes ein File OPEN ist).
Auftretende Fehler innerhalb des „ALL"-Loops werden im Zielbereich gespeichert, der Loop wird fortgesetzt, bis das letzte Unit bearbeitet ist.

**$C6: SET PREFIX**
Parmcount = 1
Pathname – 2 Byte-Adresse
Dieses COMMAND setzt das MLI-interne PREFIX und sucht das entsprechende Volume bzw. die entsprechende (Sub)directory.
Zur Definition des Pathname im PARMBLOCK siehe CREATE.
SET PREFIX ist auch mit einem Nullargument möglich, d.h. der Pathname besteht nur aus einem „/". In diesem Fall wird das PREFIX gelöscht. Ein Test, ob vorher ein PREFIX definiert war, erfolgt dabei nicht.
Neben den „allgemeinen" Fehlern kann hier noch ERR#40: INVALID PATHNAME im speziellen Kontext auftreten, nämlich entweder mit der Bedeutung „letztes File im PREFIX-Path ist keine (Sub)directory" oder mit „altes Prefix und neuer (unvollständiger) Name zusammen länger als $40 Zeichen".

**$C7: GET PREFIX**
Parmcount = 1
Zielpuffer – 2 Byte-Adresse
Das MLI-interne PREFIX wird in den Zielpuffer kopiert (und ein letzter „/" angehängt). Ist kein PREFIX gesetzt, folgt RTS ohne Fehler.
Der Zielbereich wird mit einem angenommenen Platzbedarf von $40 Bytes gegen die SBM geprüft: der einzig mögliche Fehler für dieses COMMAND ist ERR#56: BAD BUFFER ADDRESS.

**$C8: OPEN**
Parmcount = 3
Pathname – 2 Byte-Adresse
Filepuffer – 2 Byte-Adresse
Reference Number – 1 Byte Resultat
Achtung: dieses COMMAND ist nicht identisch mit dem OPEN des BASIC.SYSTEM!
Im Gegensatz zum OPEN des BASIC.SYSTEM muß ein File bereits existieren, um via OPEN angesprochen zu werden, ansonsten folgt ERR#46: FILE NOT FOUND.
In der System Bit Map wird der Filepuffer mit einem Platzbedarf von $400 Bytes ab der angegebenen Startadresse geprüft und belegt. Ist der entsprechende Platz nicht frei, folgt ERR#56: BAD BUFFER ADDRESS.
Die Reference Number (Bereich: 1-8) wird in den Parameterblock eingetragen. Über diese Refno finden alle zukünftigen Aktionen mit diesem File statt. Auch das BASIC.SYSTEM verwandelt ein „WRITE DATENFILE" o.ä. in ein „WRITE auf den File mit der Refno xx".
Außer den „allgemeinen" Fehlern kann noch ERR#50: FILE IS (ALREADY) OPEN auftreten. Hier ist die Prüfung etwas merkwürdig ausgefallen (s. „Spezielles").

**$C9: NEWLINE**
Parmcount = 3
Reference Number – 1 Byte
Enable Mask – 1 Byte
Newline Character – 1 Byte
NEWLINE kann nur auf OPEN Files angewendet werden und setzt eine End-Bedingung für READ. Wenn NEWLINE aktiv ist, d.h. dieser Befehl wurde nach dem OPEN für einen File gegeben, wird bei jedem Byte, das via READ gelesen wird, zuerst mit der „Mask" ein AND durchgeführt. Danach erfolgt ein Vergleich mit dem „Character". Stimmt das gelesene Byte (nach der Maskierung) mit dem „Character" überein, wird READ beendet. Das BASIC.SYSTEM setzt z.B. für Textfiles die „Mask" auf $7F und den „Character" auf $0D, d.h. jedes READ endet spätestens bei einem <CR> ($8D oder $0D).
Der einzig mögliche Fehler für NEWLINE ist ERR#43: INVALID REFERENCE NUMBER.

**$CA: READ**
Parmcount = 4
Reference Number – 1 Byte
Zielpuffer – 2 Byte-Adresse
Request Count – 2 Byte
Transfer Count – 2 Byte Resultat
READ kann nur auf OPEN Files angewendet werden und liest „Request Count" Bytes ab der momentanen „Position im File" in den Zielpuffer.
Vor READ wird gegen den EOF des Files geprüft und der Request Count ggf. entsprechend herabgesetzt. Bleibt dabei nichts mehr übrig, weil die „Position im File" bereits auf dem EOF steht, folgt ERR#4C: END OF FILE ENCOUNTERED.
Der Zielbereich wird gegen die SBM geprüft. Wenn nicht der gesamte Bereich frei ist, folgt ERR#56: BAD BUFFER ADDRESS.
Wird während READ ein „sparse" Block des Files erreicht, wird die entsprechende Anzahl von Bytes im Zielgebiet auf 00 gelöscht.

Nach dem Ende von READ (entweder durch Abzählen des ggf. limitierten Request Count oder durch ein NEWLINE) wird im Parameterblock der Transfer Count, also die Anzahl der tatsächlich transferierten Bytes gesetzt.
Das Setzen des Transfer Count erfolgt auch im Fall eines Abbruchs durch einen Fehler.
Während READ verschiebt sich die „Position im File" um die Anzahl der gelesenen Bytes.
READ hat für (Sub)directories ein merkwürdiges Verhalten, wenn der EOF erreicht wird (s. „Spezielles").
Außer den „allgemeinen" und den oben erwähnten Fehlern gibt es bei READ keine weiteren speziellen Fehlerbedingungen.

**$CB: WRITE**
Parmcount = 4
Reference Number – 1 Byte
Quellpuffer – 2 Byte-Adresse
Request Count – 2 Byte
Transfer Count – 2 Byte
WRITE ist nur für OPEN Files möglich und schreibt „Request Count" Bytes aus dem Quellpuffer ab der „momentanen Position" auf den entsprechenden File.
Der File wird ggf. durch Anhängen weiterer Blocks und Erhöhung des Storage Type erweitert, der EOF erhöht sich ebenfalls.
Das MLI versucht immer, „Request Count" Bytes auf den File zu schreiben. Ein Abruch kann nur durch eine Fehlerbedingung (nicht durch NEWLINE o.ä.) erfolgen.
Nach Beendigung von WRITE wird der Transfer Count im Parameterblock mit der Anzahl der tatsächlich geschriebenen Bytes gesetzt. Dieses Setzen erfolgt auch bei Abbruch durch Fehler (und hat nur da einen Sinn). Die „Position im File" verschiebt sich während WRITE um die Anzahl der geschriebenen Bytes.
Außer ERR#48: VOLUME FULL und den „allgemeinen" Fehlermöglichkeiten meldet sich WRITE mit ERR#E: FILE ACCESS ERROR, wenn versucht wird, diesen Befehl auf (Sub)directories anzuwenden.

**$CC: CLOSE**
Parmcount = 1
Reference Number – 1 Byte oder 00 („ALL")
Dieses COMMAND unterscheidet zwischen spezifizierter Refno (in diesem Fall muß das File OPEN sein, sonst folgt ERR#43: INVALID REFERENCE NUMBER) und „ALL" mit der Refno = 00 – hier wird der gesamte FCB nach OPEN Files abgesucht und alle Files werden geschlossen.
CLOSE ruft FLUSH auf (s. dort). Danach wird der Filepuffer des entsprechenden Files wieder in der SBM freigegeben und der FCB-Eintrag des Files gelöscht.
Bei einem „ALL"-Loop werden eventuelle Fehlernummern zwischengespeichert, es werden auf jeden Fall sämtliche FCB-Einträge (= OPEN Files) bearbeitet.
Außer den „allgemeinen" gibt es für CLOSE keine spezifischen Fehlermöglichkeiten.

**$CD- FLUSH**
Parmcount = 1
Reference Number – 1 Byte oder 00 („ALL")
Dieses COMMAND unterscheidet genauso wie CLOSE zwischen spezifizierter Refno und

„ALL" mit der Refno = 00 – hier wird der gesamte FCB abgearbeitet. Allerdings spielt dabei dann der LEVEL des BASIC.SYSTEM eine Rolle.
Von CLOSE aus wird FLUSH jeweils mit spezifizierter Refno für jeden einzelnen File aufgerufen.
Eventuell noch ungeschriebene Bestandteile eines Files (Index/Datenblock) werden aus dem Filepuffer auf das Volume zurückgeschrieben, der File-Eintrag wird herausgesucht und auf den neuen Stand gebracht (EOF, BLOCKS USED, Storage Type etc.). Danach folgt ein Update der Directory-Struktur.
Innerhalb eines „ALL"-Loops werden eventuell auftretende Fehlernummern zwischengespeichert. Der Loop arbeitet auf jeden Fall sämtliche FCB-Einträge (= OPEN Files) ab.
Spezielle Fehlermöglichkeiten außer den „allgemeinen" gibt es bei FLUSH nicht.

**$CE: SET MARK**
Parmcount = 2
Reference Number – 1 Byte
Position gewünscht – 3 Byte (lo/mid/hi)
Dieses COMMAND kann nur auf OPEN Files angewendet werden und setzt die „Position im File", d.h. der entsprechende Block des Files wird lokalisiert und in den Filepuffer eingelesen.
Auch wenn SET MARK MLI-intern eine zentrale Funktion hat, beschränkt sich der „reguläre" Zugriff auf dieses COMMAND von der „user"-Seite aus auf ein Setzen der Position, wobei hier vorher gegen den EOF des Files geprüft wird. Wird vom „user"-Programm eine Position spezifiziert, die hinter dem momentanen EOF des Files liegt, folgt ERR#4D: POSITION OUT OF RANGE. Dies ist auch der einzige spezifische Fehler, der bei SET MARK außer den „allgemeinen" auftreten kann.

**$CF: GET MARK**
Parmcount = 2
Reference Number – 1 Byte
Position im File – 3 Byte Resultat
Dieses COMMAND kann nur auf OPEN Files angewendet werden und kopiert die „Position jetzt im File", also den Start des nächsten READ oder WRITE in den PARMBLOCK. Außer ERR#43: INVALID REFERENCE NUMBER sind dabei keine Fehler möglich.

**$D0: SET EOF**
Parmcount = 2
Reference Number – 1 Byte
neuer EOF – 3 Byte (lo/mid/hi)
Dieses COMMAND kann nur auf OPEN Files angewendet werden und setzt den EOF eines Files.
SET EOF ist nur für Datenfiles (nicht für Subdirectories) möglich.
Ist WRITE auf den File nicht erlaubt (Bit 1 in Access gelöscht), folgt ERR#4E: FILE ACCESS ERROR.
Intern wird zwischen einer Vergrößerung des Files, bei der lediglich der EOF im FCB-Eintrag gesetzt wird, und einer Verkleinerung des Files, bei der Blocks aus dem File gelöscht werden, unterschieden.
Bei einer Vergrößerung des EOF gibt es außer ERR#43: INVALID REFERENCE NUMBER keine weiteren Fehlermöglichkeiten.

Bei einer Verkleinerung des Files werden Blocks aus dem File wieder freigegeben und innerhalb der Master/Indexblock(s) gelöscht, bis der File auf die gewünschte Länge gekürzt ist. Mit einem EOF = 00 00 00 kann ein File vollständig gelöscht werden, auch wenn er gegen DESTROY geschützt ist(!)
Bei einer Verkürzung des Files ändert sich dabei ggf. ebenfalls der Storage Type.
Spezifische Fehlermöglichkeiten außer den „allgemeinen" und FILE ACCESS ERROR gibt es für SET EOF nicht.

**$D1: GET EOF**
Parmcount = 2
Reference Number – 1 Byte
EOF – 3 Byte Resultat
Dieses COMMAND kann nur auf OPEN Files angewendet werden und kopiert den momentanen EOF des Files (FCB-Eintrag) in den PARMBLOCK. Außer ERR#43: INVALID REFERENCE NUMBER gibt es keine Fehlermöglichkeiten.

**$D2: SET BUF**
Parmcount = 2
Reference Number – 1 Byte
Pufferstart – 2 Byte-Adresse
Dieses COMMAND kann nur auf OPEN Files angewendet werden und setzt einen neuen Filepuffer. Der Inhalt des alten Filepuffers wird in den neuen Filepuffer kopiert, der alte Filepuffer wird in der SBM wieder freigegeben.
Der Bereich des neuen Filepuffers ($400 Bytes) wird gegen die SBM geprüft. Wenn er nicht vollständig frei ist, fogt ERR#56: BAD BUFFER ADDRESS. Dies ist außer ERR#43 der einzig mögliche Fehler.
Da der neue Puffer vor der Freigabe des alten Puffers geprüft wird, ist das speicherplatzmäßige Verschieben eines Filepuffers in einem Schritt nur möglich, wenn der neue Filepuffer nicht teilweise den Platz des alten Puffers belegt. Für ein Verschieben um z.B. $100 Byte sind also zwei Aufrufe von SET BUF nötig.

**$D3: GET BUF**
Parmcount = 2
Reference Number – 1 Byte
Pufferstart – 2 Byte-Adresse als Resultat
Dieses COMMAND kann nur auf OPEN Files angewendet werden und schreibt die Startadresse des entsprechenden Filepuffers in den PARMBLOCK. Der einzig mögliche Fehler ist hier ERR#43: INVALID REFERENCE NUMBER.

# Anhang F. Fehlermeldungen – „normal” und SYSTEM DEATH

## 1) „Normale” Fehlermeldungen

ERR#00 – NO ERRORS
Diese Fehlernummer wird bei fehlerfreier Ausführung eines Commands gesetzt.

ERR#01 – BAD SYSTEM CALL NUMBER
Das COMMAND-Byte wurde in der MLI-internen Tabelle nicht gefunden, d.h. es fand ein MLI-Aufruf mit einem nicht definierten COMMAND statt.

ERR#04 – BAD SYSTEM CALL PARAMETER COUNT
Die Anzahl der Parameter (erstes Byte im PARMBLOCK) stimmt nicht mit der definierten Anzahl der Parameter für dieses COMMAND überein.

ERR#25 – INTERRUPT TABLE FULL
Es können maximal vier Service-Startadressen mit ALLOC INTERRUPT in der INTERRUPT TABLE eingetragen werden. Jeder Versuch, weitere Services einzutragen, führt zu dieser Fehlermeldung.

ERR#27 – I/O ERROR
Ein „Catch all” für diverse Disk-Fehler wie z.B. unleserliche Diskette, Klappe des Laufwerks offen – aber auch „Blocknummer außerhalb des zulässigen Bereichs”.

ERR#28 – NO DEVICE CONNECTED
Wird entweder gesetzt, wenn ein nicht existierendes Unit adressiert wurde (keine definierte Startadresse der RWTB in der DRIVER TABLE) oder innerhalb der RWTB selber (z.B. bei Adressierung eines nicht vorhandenen Drive 2 an einem DISKII-Controller).

ERR#2B – DISK WRITE PROTECTED
Dieser Fehler wird nur innerhalb der DISKII-RWTB gesetzt, die RAMDisk ist nie schreibgeschützt.

ERR#2E – DISK SWITCHED AND FILE OPEN
Dieser Fehler kann nur bei ON LINE („ALL”) auftreten bzw. erkannt werden: er wird gesetzt, wenn ein Unit plötzlich einen anderen Volume-Namen hat und auf dem vorherigen Volume noch File(s) OPEN sind. Eine Veränderung der MLI-Parameter findet dabei nicht statt, d.h. dieser Fehler ist durch ein erneutes Einlegen der richtigen Disk behebbar!

ERR#40 – INVALID PATHNAME
Bedeutet in den meisten Fällen tatsächlich „ungültiger Pathname”, d.h. ungültige Zeichen innerhalb des Path, einzelne Namen zu lang etc.

Bei SET PREFIX wird dieser Fehler auch erzeugt, wenn ein Name im PREFIX-Path keine (Sub)directory ist, z.B. bei PREFIX /DEMO/BASICPROG/SUBDIR, wobei hier BASICPROG für einen Datenfile mit einem BASIC-Programm darin steht.
Innerhalb des Suchprozesses nach einem File-Eintrag (FIND FILE ENTRY) wird dieser Fehler auch gesetzt, wenn die Suche in einer Volume Directory endet, d.h. der Path nur aus einem Namen besteht. Je nach der Art des COMMAND wird dieser Fehler entweder maskiert (OPEN/SET PREFIX etc.) oder führt zum Abbruch (SET EOF, SET FILE INFO..).

ERR#42 – MAXIMUM NUMBER OF FILES OPEN
Es wurde versucht, mehr als 8 Files gleichzeitig OPEN zu halten. Der FCB hat Platz für maximal 8 Files, d.h. dieser Fehler könnte auch FILE CONTROL BLOCK FULL heißen.

ERR#43 – INVALID REFERENCE NUMBER
Dieser Fehler kann bei zwei verschiedenen Gelegenheiten auftreten:
1) Reference Number nicht im Bereich 1-4 (DEALLOC INTERRUPT) bzw. 1-8 (File Reference Number);
2) Der FCB-Eintrag mit dieser Reference Number ist nicht besetzt, d.h. es existiert kein OPEN File mit dieser Reference Number.

ERR#44 – DIRECTORY NOT FOUND
Wenn im Verlauf eines Suchprozesses (FIND FILE ENTRY) ein File nicht gefunden wird, wird hier noch einmal zusätzlich unterschieden: wenn dieser nicht gefundene File der letzte Name im Path war, wird ERR#46, ansonsten ERR#44 ausgegeben, weil das Programm davon ausgeht, daß dieser „nicht-letzte" Name im Path für eine Directory steht.

ERR#45 – VOLUME NOT FOUND
Das entsprechende Volume (erster Name im (vollständigen) Path) konnte trotz Absuchen aller Units nicht gefunden werden.

ERR#46 – FILE NOT FOUND
Der File (letzter Name im Path) konnte nicht gefunden werden (vgl. ERR#44). Diese Fehlernummer wird intern auch benutzt, um sicherzustellen, daß ein File nicht bereits existiert (CREATE und RENAME).

ERR#47 – DUPLICATE FILENAME
Der File existiert bereits (CREATE) oder ein File mit dem neuen Namen existiert bereits (RENAME). Vgl. ERR#46.

ERR#48 – VOLUME FULL
Auf dem Volume sind sämtliche Blocks belegt. Dieser Fehler kann nur bei bei einem COMMAND auftreten, das weitere Blocks belegt, also nur bei WRITE und CREATE.

ERR#49 – VOLUME DIRECTORY FULL
In der Volume Directory ist kein Platz mehr für einen weiteren File-Eintrag. Im Gegensatz zu Subdirectories kann die Volume Directory nicht durch Anhängen weiterer Blocks vergrößert werden. Dieser Fehler kann nur in CREATE erzeugt werden.

ERR#4A – INCOMPATIBLE FILE FORMAT
Die „Min-Version“ im File-Eintrag ist höher als die Version des MLI, mit der gerade versucht wird, diesen File zu bearbeiten. Diese beiden Versionsnummern werden in OPEN verglichen. Dieser Fehler steht dem Benutzer erst für zukünftige Versionen von ProDOS ins Haus, wenn Apple, Inc. sich dazu entschließt, den ProDOS-Verhau komplett zu machen.

ERR#4B – UNSUPPORTED STORAGE TYPE
Es wurde versucht, einen File mit nicht definiertem Storage Type zu erstellen (CREATE). Dieser Fehler könnte ebenfalls in Zukunft auftreten, wenn neuere MLI-Versionen noch weitere Storage Types einführen.

ERR#4C – END OF FILE ENCOUNTERED
Es wurde versucht, mit READ von einem File zu lesen, in dem die „Position jetzt“ bereits auf dem EOF steht, d.h. der File ist hier zu Ende und es gibt nichts mehr zu lesen.

ERR#4D – POSITION OUT OF RANGE
Es wurde mit SET MARK versucht, eine „Position jetzt“ im File zu setzen, die größer ist als der momentane EOF. Dieser Fehler wird nur für SET MARK-Aufrufe durch ein „user“-Programm erzeugt, bei den internen Aufrufen von SET MARK findet der Test gegen den EOF nicht statt!

ERR#4E – FILE ACCESS ERROR
Dieser Fehler ist ein weiterer „Catch all“. Er wird erzeugt, wenn versucht wurde, DESTROY/RENAME/WRITE/READ auf einen File anzuwenden, der die entsprechenden Bit(s) im Access nicht gesetzt hat. Des weiteren wird dieser Fehler gesetzt, wenn versucht wurde, eine Subdirectory, in der noch File-Einträge stehen, mit DESTROY zu löschen oder bei der versuchten Anwendung von SET EOF auf (Sub)directories.

ERR#50 – FILE IS OPEN
Diese Fehlernummer steht für zwei verschiedene Bedingungen:
1) Versuch der Anwendung eines COMMAND, das auf OPEN Files nicht anwendbar ist (z.B. RENAME, DESTROY, OPEN selber, etc.);
2) RENAME eines Volumes, auf dem File(s) OPEN sind. Das RENAME ist dabei ebenfalls nicht möglich – dieser Fehler ist hauptsächlich eine Konzession an die interne File-Verwaltung des BASIC.SYSTEM.

ERR#51 – DIRECTORY STRUCTURE DAMAGED
Innerhalb des Suchprozesses wird die Anzahl der aktiven Einträge in einer Directory heruntergezählt. Stimmt diese Anzahl nicht mit der Anzahl der gefundenen File-Einträge überein, wird dieser Fehler gesetzt. Im Normalbetrieb ist dieser Fehler praktisch unmöglich - beim Betrieb der ProDOS-Versionen 1.0.1 und 1.0.2 zusammen mit 48k RAM tritt er praktisch zwangsläufig auf (s. „Unterschiede zu 48k-Systemen“).

ERR#52 – NOT A PRODOS VOLUME
Diese Fehlernummer wird nur in ON LINE gesetzt und steht für „Unit zwar lesbar, aber Block 00 02 enthält keinen ProDOS Volume Directory Key Block“.

ERR#53 – INVALID SYSTEM CALL PARAMETER
Es wurde versucht, eine Interrupt Service-Routine mit einer Startadresse unter $100 in der INTERRUPT TABLE einzutragen (ALLOC INTERRUPT).
Allgemeiner: einer der Parameter aus dem PARMBLOCK wurde als nicht zulässig erkannt.

ERR#55: – VOLUME CONTROL BLOCK TABLE FULL
Der VCB hat Platz für maximal 8 Volume-Einträge. Da auf der System Global Page insgesamt 14 Units verzeichnet werden können, ist dieser Fehler praktisch vorprogrammiert, wenn diese Möglichkeit auch genutzt wird. (Allerdings sind APPLEs mit mehr als 8 Laufwerken wirklich unwahrscheinlich).

ERR#56: – BAD BUFFER ADDRESS
Jedes MLI-COMMAND, in dem eine Zielpuffer-Adresse angegeben ist (also nicht nur READ und READ BLOCK, sondern auch ON LINE und GET PREFIX) sowie OPEN und SET BUF (Filepuffer) und erstaunlicherweise WRITE BLOCK prüft den Zielbereich dieser Schreibaktion (WRITE BLOCK: den Quellbereich) gegen die System Bit Map. Nur bei OPEN und SET BUF wird der entsprechende Bereich auch in der SBM belegt, für alle anderen Befehle erfolgt nur ein Test.
Ist der entsprechende Bereich in der SBM nicht vollständig frei, wird ERR#56 gesetzt.

ERR#57 – DUPLICATE VOLUME
Dieser Fehler wird nur von ON LINE („ALL") gesetzt, wenn zwei Volumes mit gleichen Namen entdeckt wurden und auf mindestens einem der beiden Volumes ein File OPEN ist.

ERR#5A – FILE STRUCTURE DAMAGED
Ein sehr unwahrscheinlicher Fehler. Sollte das MLI intern versuchen, einen Block wieder freizugeben (DEALLOCATE BLOCK), dessen Blocknummer die Maximalzahl der möglichen Blocks (TOTAL BLOCKS) auf dem entsprechenden Volume überschreitet, wird dieser Fehler gesetzt. Beim Belegen eines Blocks wird dagegen nicht MLI-intern geprüft, ob die Blocknummer im zulässigen Bereich ist. Hier meldet sich z.B. die DISKII-RWTB mit einem I/O ERROR. Allerdings muß dazu vorher jemand in der VBM herumgebastelt haben.

## 2) Fehlermeldungen von SYSTEM DEATH

ERR#00
Es erfolgt keine Fehlermeldung, d.h. keine Anzeige von „ERR-xx"
Hierzu gibt es zwar einen Einsprung, zu dem auch ein Vektor in der System Global Page steht (s. „System Global Page") – dieser Vektor ist allerdings nach erfolgtem Relocate gelöscht, d.h. im „Normalbetrieb" findet dieser Aufruf nie statt.

ERR#01 – INTERRUPT FROM UNKNOWN SOURCE (Version 1.0.1)
Es hat ein IRQ stattgefunden, ohne daß sich eine der Service-Routinen dafür zuständig fühlt bzw. es hat ein IRQ stattgefunden, ohne daß eine Service-Routine installiert ist. Nach dem Absuchen der gesamten INTERRUPT TABLE und dem Aufrufen gefundener Services endet der IRQ-Handler ($D13A) mit diesem Fehler.
Dies ist der einzige logisch begründbare SYSTEM DEATH.

ERR#02 – INTERRUPT FROM UNKNOWN SOURCE (Version 1.0.2)
In der Version 1.0.2 wurde die ERR#01 auf 02 geändert.

ERR#0A – NO VCB ENTRY FOR OPEN FILE
Bei der Bearbeitung eines Files stellt sich plötzlich heraus, daß für diesen File keine VCB-Eintrag, d.h. kein Unit existiert – der File wurde aus dem Nichts heraus eröffnet (?). Trotz hartnäckiger Versuche gelang es nur mit direktem Herumpoken im MLI, diesen Fehler zu erzeugen.

ERR#0B – NO ASSIGNED FILEBUFFER FOR OPEN FILE
Bei der Bearbeitung eines Files stellt sich plötzlich heraus, daß der Filepuffer verschwunden ist, d.h. beim CLOSE eines anderen Files oder einem SET BUF wurde die Startadresse des Filepuffers in der BUFFER TABLE „aus Versehen" mit gelöscht (?).
Hier gilt dasselbe wie für ERR#0A.

ERR#0C – NON EXISTING BLOCK IN FILEBUFFER
Nachdem SET MARK einen existierenden Block innerhalb eines Files lesen will, stellt sich beim Setzen der Blocknummer plötzlich heraus, daß dieser Block aus dem FCB-Eintrag verschwunden ist. Anstelle des „Blocks jetzt im Speicher" bzw. des „Indexblocks jetzt im Speicher" steht hier die Blocknummer 00 00.
Die einzige Gelegenheit, bei der die Datenblocknummer auf 00 00 gesetzt wird, ist READ für Directories, wenn dabei der EOF überschritten wird – und jeder weitere Aufruf von READ bricht vorher mit EOF ENCOUNTERED ab.

ERR#40/50/60.... – TRIED TO REMOVE BLOCKS FROM UNDEFINED STORAGE TYPE
Wenn sich die Programmausführung bei einem DESTROY oder SET EOF um die diversen Prüfungen des Storage Type herumgetrickst hat und schlußendlich in REMOVE BLOCKS mit einem File mit Storage Type <> 1/2/3 landet – dann ist das Programm so beleidigt, daß es sich mit dem Storage Type * 16 als Fehlernummer aufhängt.

# Anhang G. Glossar der verwendeten Abkürzungen

AUX — Auxilliary RAM – zusätzliche 64k der Extended 80-Col Card, hier hauptsächlich als Unterscheidung zu MAIN („normaler" RAM-Bereich des APPLE) verwendet.

Blockno — Blocknummer

DIR — Directory – Inhaltsverzeichnis

EOF — End of File – steht für die höchstmögliche „Position im File", d.h. über diese Position hinaus kann nichts mehr von dem File gelesen werden, weil der File hier zu Ende ist. In manchen englischen Manuals steht hierfür auch der verwirrende Ausdruck „reading unwritten data".

FCB — File Control Block – ein Speicherbereich des MLI, in dem maximal 8 (OPEN) Files festgehalten werden können. Wenn ein File via OPEN eröffnet wird, wird ein FCB-Eintrag erstellt. Er enthält unter anderem die Reference Number des Files, die „Position im File", von der als nächstes gelesen/geschrieben wird und den EOF des Files.

MLI — Machine Language Interface – Oberbegriff für sämtliche hier besprochenen Routinen. Ob der Begriff ProDOS für das MLI und das BASIC.SYSTEM zusammen steht (oder nur für das MLI), ist eine mehr philosophische Frage.

Refno — Reference Number – korrekterweise müßte zwischen „File Reference Number" und „Interrupt Reference Number" unterschieden werden. Da aber die Verwendung der „Interrupt Reference Number" auf die Funktion DEALLOC INT beschränkt bleibt, ist „File" meistens entfallen.

RWTB — Read/Write Track/Block – ein von mir geschaffener Begriff der Routinen zum Lesen/Schreiben eines Blocks von/auf ein Volume. Der Begriff wurde in Anlehnung an die RWTS (Read/Write Track/Sector) von DOS 3.3 gewählt.

SBM — System Bit Map – nicht zu verwechseln mit der VBM. Die System Bit Map existiert innerhalb des Speichers und steht für die Belegung des RAM durch das MLI, die VBM steht auf dem Volume und gibt die Belegung der Blocks des Volume wieder.

Sectorno — Sektornummer, innerhalb der Listings aus Platzgründen auch manchmal als „Secno" noch weiter abgekürzt.

SGP — System Global Page – innerhalb dieses Buchs inkonsequent verwendet. Apple, Inc. unterscheidet zwischen der „Global Page" des BA-

| | |
|---|---|
| | SIC.SYSTEM auf $BE00 und der „System Global Page" des MLI auf $BF00. Da das BASIC.SYSTEM hier nicht behandelt wird, fehlt manchmal das „System". |
| LC | Language Card – bzw. der schaltbare Bereich $D000..$FFFF. Der Terminus „LC-Bereich" steht auch für die entsprechenden Adressen im AUX-RAM. |
| VBM | Volume Bit Map – ein oder mehrere Blocks auf einem Volume, in dem die freien/belegten Blocks des Volume Bit-weise verzeichnet sind. |
| VCB | Volume Control Block – ein Speicherbereich des MLI, in dem maximal 8 verschiedene Volumes festgehalten werden können. Ein Eintrag innerhalb des VCB umfaßt den Volume-Namen, Unitnummer und diverse Daten über das Volume selber (Anzahl der freien Blocks etc.). |
| VolDIR | Volume Directory |

# INDEX

Dieser Index enthält (im Gegensatz zu sonstigen Gewohnheiten) nicht jeweils die erste Stelle, an der ein Begriff oder ein Name auftaucht, sondern verweist (soweit möglich) auf die Stelle, wo der entsprechende Begriff erklärt bzw. besprochen wird. Er ist deshalb manuell erstellt worden und mit Sicherheit nicht vollständig.

Falls Sie bereits einmal in einem anderen Handbuch einen Index im Stile von:

...
Slot 60,72,85...
...

und auf den entsprechenden Seiten dann Sätze wie:

... kann über jeden Slot des Apple ...
... geht auch über einen anderen Slot als 6 ...

gefunden haben (anstelle einer gesuchten Erklärung dieses Begriffs), dann werden Sie hoffentlich die Vorteile dieses Index zu schätzen wissen.

Die Namen einzelner Routinen und Kommandos innerhalb des Programmes PRODOS wurden mit aufgenommen. Folgende Abkürzungen wurden verwendet:
Die Namen einzelner Routinen und Kommandos innerhalb des Programmes PRODOS wurden mit aufgenommen. Folgende Abkürzungen wurden verwendet:

CD = Command bzw. Start einer Kommando-Routine
SR = Subroutine
EP = Einsprungpunkt bzw. logischer Knotenpunkt.

Bei MLI-Kommandos steht der erste Verweis normalerweise für die Fundstelle im Abschnitt „Logische Abläufe".

80 Zeichen-Karte (80 Col Card) 283ff
  Test im Relocator 200
  Bankswitching 284ff
  neue Version 308
    Test im Relocator 311, 314, 323
80STORE Softswitch 285

A
Abschaltverzögerung des Laufwerkmotors 251
ACCESS 11
ACTIVATE (SR) 131, 391
Address Field
  Inhalt/Aufteilung 238
  Header/Trailer 238
  Lesen/Auswertung
    Boot-Programm 182
    Urlader 188
    RWTB 264, 276
ALLOC/DEALLOC INT (CD) 34, 74
ALLOC INTERRUPT-Kommando 74, 335, 450
ALLOCATE NEXT BLOCK (SR) 15, 46, 104, 364
APPEND BLOCK TO SUBDIRECTORY (EP) 39, 85, 345
ALTZP Softswitch 284
Arm (Diskettenlaufwerk) 236
ARM MOVER
  EP - Urlader 186
  SR - RWTB 163, 273
  DELAY (SR - RWTB) 275
Autosyncs 255
AUX-INFO 155, 418
AUX MEMORY (80-Z-Karte) 283ff, 464
AUXMOVE (SR) 293

B
BANKSEL (SGP) 230
BANKSEL1 (SGP) 230
BLOCK IN MEMORY (EP) 112, 372
Blockno, Blocknummer 464
BLOCKREAD/WRITE
  Innerhalb des Filepuffers 114, 374
  -Kommandos 74, 334
BLOCKS USED 12, 442
Boot-ROM 249
BUBIT (SGP) 228, 431
BUFTBL (SGP) 226
Buffer Index 11, 149ff

C
CALCFREE (SR) 102, 362
CLEAR PREFIX 23

CLOSE-Kommando 15, 28, 456
Blockbeschreibung 58
Listing 131, 392
Error Handler 135, 395
COMMAND
$00 (RWTB) 272
COMMAND-Byte 16
Command-Handler:
$4x: 17, 335
$8x: 17, 73, 74, 333, 334
$C0-$D3: 17, 35, 73, 77, 333, 338
Controller-Karte 249
COPY&VERIFY PATHNAME (SR) 36, 78, 338
CORRECT DRIVE ON (EP - RWTB) 260
CORRECT DRIVE SELECTED (EP - RWTB) 260, 270
CREATE-Kommando 5, 19, 451
Blockbeschreibung 39
Listing 85, 345

D
Data Field
Inhalt/Aufteilung 238
Header/Trailer 238
Lesen/Auswertung
Boot-Programm 182
Urlader 188
RWTB 276
DATE (SGP) 227
DEALLOCATE BLOCK (SR) 46, 103, 363
DEALLOC INT-Kommando 450
Listing 75, 335
DEALLOCATE FILE BUFFER (SR) 69, 150, 411
DEMO 4, 437
DENIBBLE (SR - Urlader) 189
DESTROY-Kommando 6, 19, 452
Blockbeschreibung 64
Listing 143, 403
DEVICE TABLE 164
DEVNUM (SGP) 164, 224
DEVTBL (SGP) 224
DIR 464
Directory 439f
Bereich löschen 86, 346
Block absuchen 94, 354
Block schreiben s. ENTRY&UPDATE
Prüfung auf ProDOS (SR) 90, 99, 350, 359
Disk, Diskette 236
„Disk"-Bytes 242
DISKID 167
DO DISK I/O (SR) 34, 74, 334
DOS 3.3 13, 244ff
Drive 236
DRV1TBL (SGP) 223
DRV2TBL (SGP) 224

E
ENTRY&UPDATE (SR) 15, 40, 89ff, 349ff
EOF 464
im File-Info 155, 442
ERROR
(EP - Urlader) 186
(EP - Relocator) 192, 194
(EP - REBOOT) 211
(EP - RAMDisk-Interface) 287
(EP - Relocator 1.1.1) 315
EXIT (EP)
MLI 33, 333
RAMDisk-Interface 287
RAMDisk-RWTB 291, 301
EXOR-Kette 246ff

F
FCB 8, 464
Absuche (SCAN FCB) 118, 379
Änderung (V1.1.1) 307
Eintrag 162, 415
Index 156, 418
FILE ALTERED (Flag)
Setzen nach WRITE/SET EOF 145, 405
File-Eintrag 441f
Berechnung der Adresse (SR) 88, 348
Erstellen (EP) 87, 347
Suchen 92f, 352
Schreiben 89, 349
File-Info 5, 10, 89, 93
Filepuffer
Pointer setzen 122, 382
Puffer setzen 149, 410
Restore der Pointer nach READ 123, 384
File Reference Number 10
File Type 155, 417
FIND&LOAD SYSTEM PROGRAM (EP - Relocator) 174, 193, 316
FIND FILE ENTRY (SR) 41, 91, 165, 351
FLUSH-Kommando 15, 28, 456
Blockbeschreibung 59
Listing 132, 393
Error Handler 135, 395
FORMAT-Kommando (RAMDisk) 285, 288, 297
Formatieren 236
FULL NAME (Flag) 160, 422

G
Gap1, 2, 3 239
GET BUF-Kommando 32, 458
Blockbeschreibung 70
Listing 152, 412
GET EOF-Kommando 32, 458
Blockbeschreibung 62
Listing 138, 398
GET FILE (SR) 8, 41, 91, 351

GET FILE INFO-Kommando 21, 453
Blockbeschreibung 62
Listing 138, 399
GET MARK-Kommando 31, 457
Blockbeschreibung 48
Listing 108, 368
GET PREFIX-Kommando 24, 454
Blockbeschreibung 37
Listing 81, 341
GET PREFIX (EP - REBOOT) 208
GET PROGRAM NAME (EP - REBOOT) 209
GET STATUS (SR) 60, 135, 395
GET STORAGE TYPE (SR) 125, 385
GET SYSTEM PREFIX (EP - REBOOT) 207
GET VBM (SR) 47, 106, 366
GET VOLDIR/PREFIX KEY (SR) 43, 95, 355
GO READ (SR - Urlader) 170, 185
GOT NAME OF PROGRAM (EP - REBOOT) 210
GOT PREFIX NAME (EP - REBOOT) 209

**H**
Handler: s. Command-Handler

**I**
I/O-Adressen 249
Indexloch 237
INITFLAG (RAMDisk) 291
INTADDR (SGP) 227
INTAKKU (SGP) 225
INTAREG, XREG, YREG, SREG, PREG (SGP) 227
INTBANK 215
INTBANK1 (SGP) 227
INTBANK2 (SGP) 225
INTENTRY (SGP/EP) 230
INTERRUPT Service
Handler 35, 425
Blockbeschreibung 35, 330
Listing 75, 216, 336
Probleme der Versionen 1.0.x 430
INTEXIT (EP) 215, 230
INTLC (SGP) 230
INTRUPTS (SGP) 226
IRQ 215
Setzen des Vektors (V1.1.1) 313
Vektor in AUX 423
IS MOTOR ON? (SR - RWTB) 266, 278
IS SAME SLOT? (SR - RWB) 268, 282

**K**
KALTSTART (EP - Monitor II/+/e) 180

**L**
LANGIRQ
(EP) 217
(SGP) 225
LANGINTEXIT (EP) 217
LASTDEV (SGP) 224
LC 464
LEVEL 12
(SGP) 228, 431
LOAD DIRECTORY (SR - Urlader) 184

**M**
MACHID
(SGP) 228
Setzen im Relocator 190ff, 431
MAIN MEMORY (80-Z-Karte) 283ff
MLI 464
Änderungen V1.0.2 304
Einsprung 32, 72, 330
Variablen 154ff, 415ff
MLIACTIVE (SGP) 229
MLIENTI (SGP/EP) 229
MLIENTRY (SGP) 222
MLIEXIT (SGP/EP) 229
MLIRTS (SGP) 229
„Monitor"-IRQ 214ff
Motor-on Delay 251
MOVER (SR - Relocator) 177ff, 201ff, 324ff
MYSEEK (SR - RWTB) 261, 272
Mystery Flag für READ 159, 421, 431
Mystery Routine 426f

**N**
NEWLINE-Kommando 25, 455
Blockbeschreibung 62
Listing 138, 399
NEWLINE-Bytes 159, 421
NEWLINEs im FCB-Eintrag 11
NEXT SLOT (EP - Relocator) 197, 321
NMI 214
NO DEVICE (SR) 33, 73, 334

**O**
ON CORRECT TRACK
EP - Urlader 187
EP - RWTB 261, 272
ONLINE-Kommando 22, 163, 454
Blockbeschreibung 38
Listing 82, 343
Error Handler 84, 344
OPEN-Kommando 24, 307, 431, 455
Blockbeschreibung 51
Listing 116, 377
für Subdirectories (EP) 118, 378

**P**
Pathname 165
Berechnung des Index im Path 95, 356
Suche nach einem Namen 94, 354
Test auf Ende des Path 96, 355f
Überspringen eines Namens 95, 355

PREFIX 9, 165
(SGP) 228
DIRECTORY 157, 420
System Prefix 161
UNIT 157, 419
PREFIX SET (EP - REBOOT) 209
Prenibble-Prozeß 243ff
PRENIBBLE&SET WRITE ADDRS (SR - RWTB) 268, 281
PRINT MESSAGE (SR - REBOOT) 211
PRODOS INSTALLIERT (EP - Relocator) 311, 315
PRODOS RELOCATED (EP - Relocator) 191
PUFFER FÜR PRENIBBLE (RWTB) 275

**R**
RAMDisk
Installation (Relocator) 179, 205, 292, 329
RWTB 163
RAMRD/RAMWRT Softswitches 284
RDADDR (SR - RWTB) 243, 264, 276
RDTRANS-Tabellen 247, 273
READ-Kommando 25, 455
Blockbeschreibung 53
Listing 119, 379
für nicht-File Typen 124, 384, 431
Stufen 1, 2, 3 120ff, 380ff
READ/WRITE
eines Sektors (EP - RWTB) 270
eines Blocks (EP - RAMDisk-Interface) 288, 298
READ Address Field
EP - Urlader 188
SR - RWTB 276
READ Data Field
EP - Urlader 188
SR - RWTB 265, 276ff
READ BLOCK-Kommando 450
Blockbeschreibung 17
Listing 74, 334
READ BLOCK (SR) 48, 107, 368
Urlader 171, 186
in den Datenteil des Filepuffers 114, 374
in den Indexteil des Filepuffers 114, 375
READ MOVER LOOP (EP) 55, 122, 383
READ TRANSLATION TABLE I/II 263
READ SECTOR
(SR - Urlader) 171
(SR - RWTB) 259
READ VBM (SR) 47, 106, 367
REBOOT (SGP) 222
REBOOT-Kommando 16, 450, 207ff
Blockbeschreibung 70
Listing 152, 413, 207ff
RECALIBRATE (EP - RWTB) 271
Refno 464
REMOVE BLOCKS (SR) 66ff, 145ff, 406ff, 433
RENAME-Kommando 20, 452
Blockbeschreibung 63
Listing 140, 400
Request Count 12
RESET 214
Handler 214f
RETRY READ (EP - RWTB) 271
REWRITE
CURRENT DATA BLOCK (SR) 116, 376
CURRENT INDEX BLOCK (SR) 116, 376
VBM 106, 366
ROMBIT
(SGP) 228
Setzen im Relocator 200
RWTB 464
Änderungen V1.0.2 303
Variablen 275

**S**
SBITMAP (SGP) 225
SBM 464
SCAN FCB (SR) 118, 379
Schreib/Lesekop 236
schreibgeschützt 256
Schrittmotor 236, 252
SEARCH SLOTS (SR - Relocator) 175f
SEARCH VOLUME/PREFIX DIR (SR) 43, 96, 356
Sectorno 464
SEEKABS (SR - RWTB) 252, 262, 272
Sektoren
Aufteilung in 237ff
Soft/Hardsektoriert 237
Skew 239
SET ARM POSITION IN MEM (EP - RWTB) 266
SET BUF-Kommando 32, 458
Blockbeschreibung 70
Listing 152, 413
SET FILE DATA (SR) 37, 81, 342
SET FILE INFO-Kommando 21, 453
Blockbeschreibung 62
Listing 139, 400
SET EOF-Kommando 31, 307, 457
Blockbeschreibung 60
Listing 135, 396
Verkürzung des Files 136ff, 397ff
SET MARK-Kommando 12, 29, 457
Blockbeschreibung 48
Listing 108, , 368
In Directories 113, 373
SET PREFIX-Kommando 23, 454
Blockbeschreibung 36
Listing 80, 340
SET SYSERR (SR) 35, 76, 337
SET-SYSERR (SGP) 223

SET TIME-Kommando 16, 19, 161, 219, 451
Änderungen (V1.1.1) 307, 415
für 48k-Systeme 233
SETTIME (SGP) 222
SETTRACKII (EP - RWTB) 261, 271
SGP 464
single density 243
Skew s. Sektor-Skew
SOS 7
SPACE ON VOLUME? (SR) 45, 101, 361
STATUS 11
Reset der Erweiterungsbits 113, 373
GET STATUS (SR) 60, 135
Stepper-Motor s. Schrittmotor
Storage Type 439
Erweiterung in WRITE 127, 130, 390
Herabsetzung in REMOVE BLOCKS 148, 409
SUBDIR 4
Subdirectory
Key Block 441
Erstellung 86, 347
Prüfung auf ProDOS (SR) 99, 359
SYSDEATH (SGP) 223
SYSDEATH2 (SGP/EP) 230
SYSERR (SGP) 223
System Bit Map
Test gegen SBM (SR) 151, 412
SYSTEM DEATH (EP) 35, 77, 337
Text im MLI 154, 414

**T**
Tabellen des MLI 71, 153, 414
TIME (SGP) 227
TITLE PAGE (SR - Relocator) 175, 196, 320
TOTAL BLOCKS 440
Track 236
Transfer Count
Setzen (SR) 121, 382
TRY TRACK (EP - RWTB) 260, 271

**U**
Uhrenkarte
Test im Relocator 199, 323
Units 164
Kopie aller Units 98, 358
Urlader 169, 183

**V**
VBM 7, 416, 465
„altered" (Flag) 158, 420
Berechnung der max. möglichen Zahl (SR) 102, 363
„Extent" (Speicherstelle) 158, 420
Laden des nächsthöheren Extent (EP) 105, 365
Nummer des ersten Blocks 155, 417
„Unit" (Speicherstelle) 158, 420
RAMDisk-VBM 285, 291, 301, 425
Zurückschreiben (REWRITE) 106, 366
VCB 7, 161, 416, 465
Absuchen nach Volume-Namen (SR) 96, 100, 360
Änderungen (V1.1.1) 307
Eintrag erstellen 99, 359
Index 156, 418
Suche nach freiem Eintrag (SR) 98, 358
Vergleich VolDIR-VCB-Eintrag 100, 360
Vektoren für NMI/RESET/IRQ 213
Setzen des IRQ-Vektors (V1.1.1) 313
VERIFY APPLEII (SR - Relocator) 177, 201, 324
VERSION (Speicherstelle) 152
Prüfung bei OPEN 117, 377
VolDIR 464
Volume Directory Key Block 5

**W**
Warmstart (nach RESET) 167
WHAT MACHINE?(SR - Relocator) 195, 311, 319
WRITE-Kommando 12, 26, 456
Blockbeschreibung 56
Listing 126, 387
MOVER LOOP (SR) 128, 388
Transfer-Loop
(EP - MLI) 127, 387
(EP - RAMDisk-Interface) 287, 296
WRITE BLOCK-Kommando 17, 451
Blockbeschreibung 34
Listing 74, 334
WRITE BLOCK (SR) 48, 107, 367
WRITE DATA FIELD (SR - RWTB) 266, 279ff
WRITE PROTECTED 256
WRITE SINGLE BYTE (SR - RWTB) 267, 281
WRITE VBM BLOCK (SR) 48, 107, 367
WRTRANS-Tabelle 245, 263, 275

**X**
XFER (SR) 294
XSAVE (SGP) 229

**Y**
YSAVE (SGP) 229

**Z**
Zwischenspeicherung des Stack - s. IRQ

## Apple ProDOS für Aufsteiger, Band 1

**Mit ausführlichen Programmbeispielen**

**von U. Stiehl**

1984, 208 S., kart., DM 28,–
ISBN 3-7785-1027-4

Begleitdiskette zu „Apple ProDOS für Aufsteiger",
DM 28,–
ISBN 3-7785-1032-0

„Apple ProDOS für Aufsteiger, Band 2",
208 S., kart., DM 28,–,
ISBN 3-7785-1036-3
Begleitdiskette, DM 28,–, ISBN 3-7785-1040-1

ProDOS ist das neue Betriebssystem für den Apple II Plus/IIe. Applesoft-Programmierer, die unter DOS 3.3 gearbeitet haben, werden sich schnell an ProDOS gewöhnen, da ProDOS und DOS 3.3 in dieser Hinsicht weitgehend kompatibel sind. Dagegen müssen Assembler-Programmierer völlig umdenken. Deshalb liegt das Schwergewicht von „Apple ProDOS für Aufsteiger, Band 1" auf der Assemblerprogrammierung und der minutiösen Darstellung der ProDOS-internen Systemadressen, die jedoch auch für Applesoft-Programmierer von großer Bedeutung sind.
Zunächst wird zunächst ein allgemeiner Überblick über das neue „Professional Disk Operation System" gegeben. Im Anschluß daran folgt eine Gegenüberstellung der Geschwindigkeit des Diskettenzugriffs bei DOS und ProDOS. Dann wird die interne Speicherorganisation detailliert beschrieben (Boot-Vorgang, Zero-Page, ProDOS-Vektoren, Basic-System-Puffer, Basic-System-Global-Page, Basic-Command-Handler, I/O-Vektoren, ProDOS-Global-Page, Language-Card-Organisation, Interrupt, Disk-Driver, Reboot-Programm usw.). Ebenso ausführlich wird die externe Speicherorganisation geschildert (Spuren, Sektoren, Blocks, Directory-Struktur, Volume Bit Map, Dateistrukturen usw.). Schließlich wird das MLI (Machine Language Interface) mit zahlreichen praktischen Anwendungsbeispielen erläutert. Insgesamt enthält ProDOS-Buch ca. 70 Seiten mit eigens für dieses Werk entwickelten Programmen. Danach wird das Basic-System technisch für fortgeschrittene Applesoft-Programmierer kurz erläutert.
„ProDOS für Aufsteiger, Band 2" beschreibt ausführlich die Anwendung des ProDOS-BASIC-SYSTEM für Applesoft-Programmierer und enthält darüber hinaus zahlreiche größere Assembler-Anwenderprogramme, die aus Platzgründen in dem ersten Band nicht mehr untergebracht werden konnten.

„Die ProDOS Analyse" ist die umfangreichste und detaillierteste Darstellung, die jemals ein Apple-Betriebssystem erfahren hat. Wer die „Innereien" von ProDOS bis zum letzten Byte, z. T. bis ins letzte Bit kennenlernen möchte, braucht dieses Buch. Das komplette Betriebssystem (Urlader, MLI, Disk-Driver, RAM-Disk-Driver und Uhr-Routine) mit Ausnahme des BASIC-SYSTEM wird mit umfangreichen Kommentaren und Übersichtstabellen disassembliert. Dabei werden alle bisherigen Versionen von 1.0.1 bis 1.1.1 berücksichtigt. „Die ProDOS-Analyse" beschreibt erstmals auch mehrere Programmierfehler, die bis in die neueste Version zu finden sind. Auch die nicht im „Technical Reference Manual" aufgeführten Eigenschaften von ProDOS werden analysiert und beschrieben, z. B. die vertrackten eingebauten Testroutinen zur Identifikation der verschiedenen Apple II Modelle und eventueller Nachbaugeräte. Programmierer, die ProDOS versionsabhängig „patchen" möchten, erhalten hier den genauen Überblick, wo was geändert werden muß, damit dies keine negativen Konsequenzen hat. Durch die minutiöse theoretische Sezierung von ProDOS eröffnen sich völlig neue programmierpraktische Perspektiven.